Benutzerhandbuch

Petri-Netz Editor

Version 1.0

Betreuer:

Dr. Wolfgang Kastner



Wien, am 1. Juni 2004

Inhalt

1. Einleitung

2. Petri-Netz Editor im Überblick
2.1. Komponenten des Petri-Netz Editors
2.2. Kontextmenü
2.3. Menü
2.4. Werkzeugleisten
2.5. Zeichenfläche
2.6. Statuszeile
2.7. Einstellungen
2.8. Druckvorschau und Drucken
2.9. Lampen

3. Entwurfsmodus
3.1. Auswahl-/Selektionsmodus
3.2. Stelle einfügen
3.3. Transition einfügen
3.4. Spracherkenner einfügen
3.5. Verbindung ziehen
3.6. Beschriftung einfügen
3.7. Lampe hinzufügen und verknüpfen

4. Simulationsmodus
4.1. Schrittweise Ausführung
4.2. Vollautomatische Ausführung

5. Beispiele
5.1. Beispiel 1 – Minimum
5.1.1. Entwerfen des Petri-Netzes
5.1.2. Testen des Petri-Netzes
5.2. Beispiel 2 - Maximum
5.2.1. Entwerfen des Petri-Netzes
5.2.2. Testen des Petri-Netzes
5.3. Beispiel 3 - Multiplikation
5.1.1. Entwerfen des Petri-Netzes
5.3.2. Testen des Petri-Netzes

6. Anhang
6.1. Sprachausgabe
6.1.1. XML Schema: SAPI
6.1.2. Petri-Netz Erweiterungen für die Sprachsynthese
6.1.3. Beispiele
6.1.3.1. Minimum-Funktion mit Sprachausgabe
6.1.3.2. Ein Petri-Netz lernt „singen“
6.2. Spracheingabe
6.2.1. XML Schema: Grammar
6.2.2. Petri-Netz Erweiterungen für die Spracherkennung
6.2.3. Beispiele
6.2.3.1. Lampensteuerung mit Spracheingabe
6.2.3.2. Minimum-Funktion mit Sprachein-/ausgabe

6.3. Petri-Netz API
6.3.1. Klassen-Übersicht
6.3.2. PNEngine und PNEngineListener
6.3.3. Beispiele
6.3.3.1. Lampensteuerung mit Spracheingabe
6.3.3.2. Entscheidungsnetz für Hebelarm
6.3.4. Beschreibung der Klassen und Schnittstellen
6.3.4.1. PNEngine
6.3.4.2. PNEngineListener
6.3.4.3. PNElement
6.3.4.4. PNStelle
6.3.4.5. PNVerbindung
6.3.4.6. PNTransition
6.3.4.7. PNSpracherkenner
6.3.4.8. PNBeschreibung
6.4. Mathematische Grundlagen
6.5. Abbildungen
6.6. Literatur

1. Einleitung

Der Petri-Netz Editor – kurz PED - ist im Laufe einer Praktikumsarbeit im Jahre 2004 von mir als eigenständige Visual Basic™-Anwendung entwickelt worden.

Die theoretischen Grundlagen des PED stammen aus dem Buch *Prozeßautomatisierung* von Gerhard-Helge Schildt/Wolfgang Kastner, erschienen im Springer Verlag in der Reihe Lehrbuch Technik [PROZAUT]. Die Definitionen im Kapitel 6.4 *Mathematische Grundlagen* sind diesem Buch entnommen.

Dieses Benutzerhandbuch bietet keine allgemeine Einführung in die Oberfläche von Windows, sondern konzentriert sich ausschließlich auf die Funktionen des Petri-Netz Editors. Dabei sollte ein Großteil der Bedienung intuitiv verständlich, und - mit einer gewissen spielerischen Herangehensweise - auch ohne diese Lektüre möglich sein.

"Der einzige Weg, der zum Wissen führt, ist Tätigkeit."
(George Bernard Shaw)

2. Petri-Netz Editor im Überblick

Über das Menü *Datei* → *Neu* → *Petri-Netz* wird ein neuer Petri-Netz Editor gestartet.

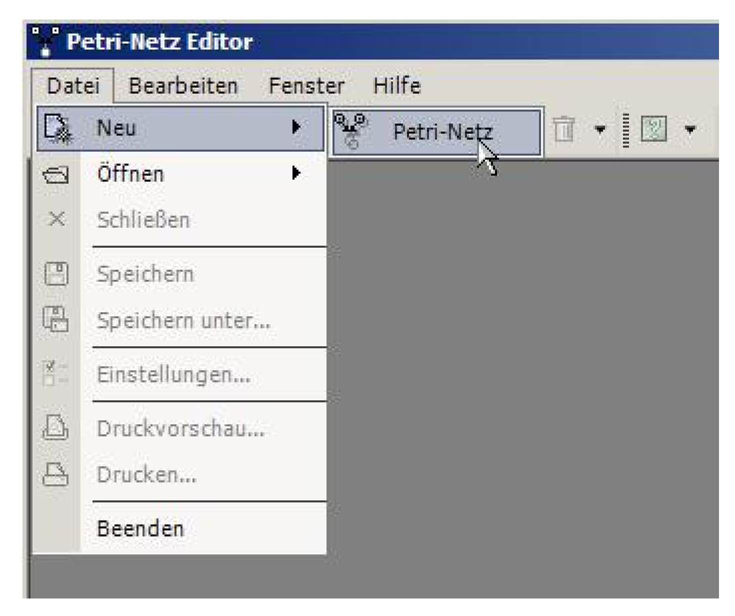


Abb. 1: Petri-Netz Editor starten

2.1. Komponenten des Petri-Netz Editors

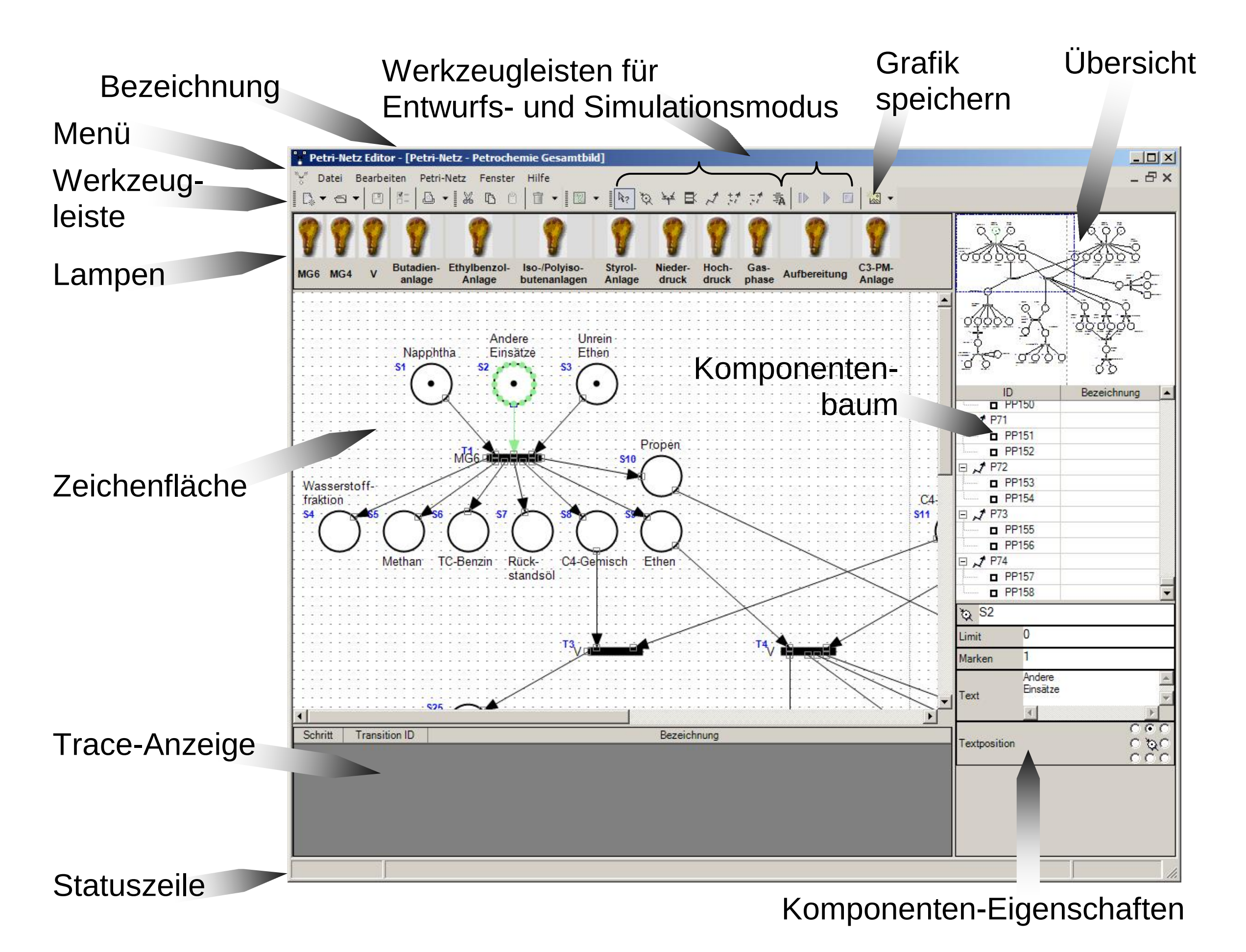


Abb. 2: Komponenten des Petri-Netz Editors

2.2. Kontextmenü

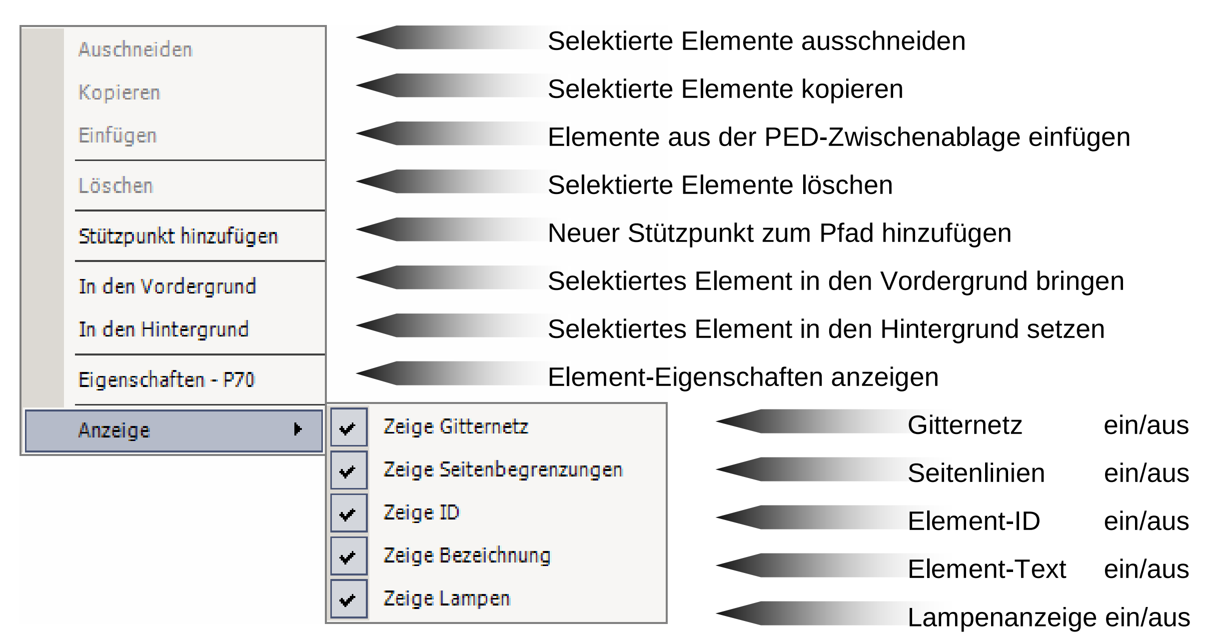


Abb. 3: Kontextmenü

Die Funktionen *Ausschneiden*, *Kopieren*, *Einfügen* und *Löschen* implementieren eine einfache Zwischenablage. Diese Zwischenablage steht zwar allen Petri-Netz Editoren zur Verfügung, ist jedoch von der systemweiten Zwischenablage zu unterscheiden.

Mit *Stützpunkt hinzufügen* wird einem Pfad ein weiterer Punkt hinzugefügt. Der Stützpunkt wird an der Stelle erzeugt, an der das Kontextmenu aufgerufen wurde.

Mit den Funktionen *In den Vordergrund* und *In den Hintergrund* wird ein Element hinsichtlich der Z-Achse verschoben. Ein Element im Vordergrund verdeckt dabei Elemente, die sich im Hintergrund befinden.

Die Funktion *Eigenschaften - ID* zeigt die zum Element gehörenden Eigenschaften an. Näheres zu den Eigenschaften findet man in Kapitel 3. *Entwurfsmodus*.

Hinweis:

Die Funktion *Stützpunkt hinzufügen* ist nur dann aktiv, wenn das Kontextmenü über einem Pfad aufgerufen wird. Genaueres dazu im Kapitel 3.5. *Verbindung ziehen*.

Die Funktionen *In den Vordergrund*, *In den Hintergrund* und *Eigenschaften – ID* sind nur dann aktiv, wenn das Kontextmenü über einem grafischen Element aufgerufen wird.

Die Funktionen *Ausschneiden, Kopieren* und *Löschen* sind nur dann aktiv, wenn Elemente selektiert sind.

Die Funktion *Einfügen* ist nur dann aktiv, wenn sich Elemente in der PED-Zwischen-ablage befinden.

2.3. Menü

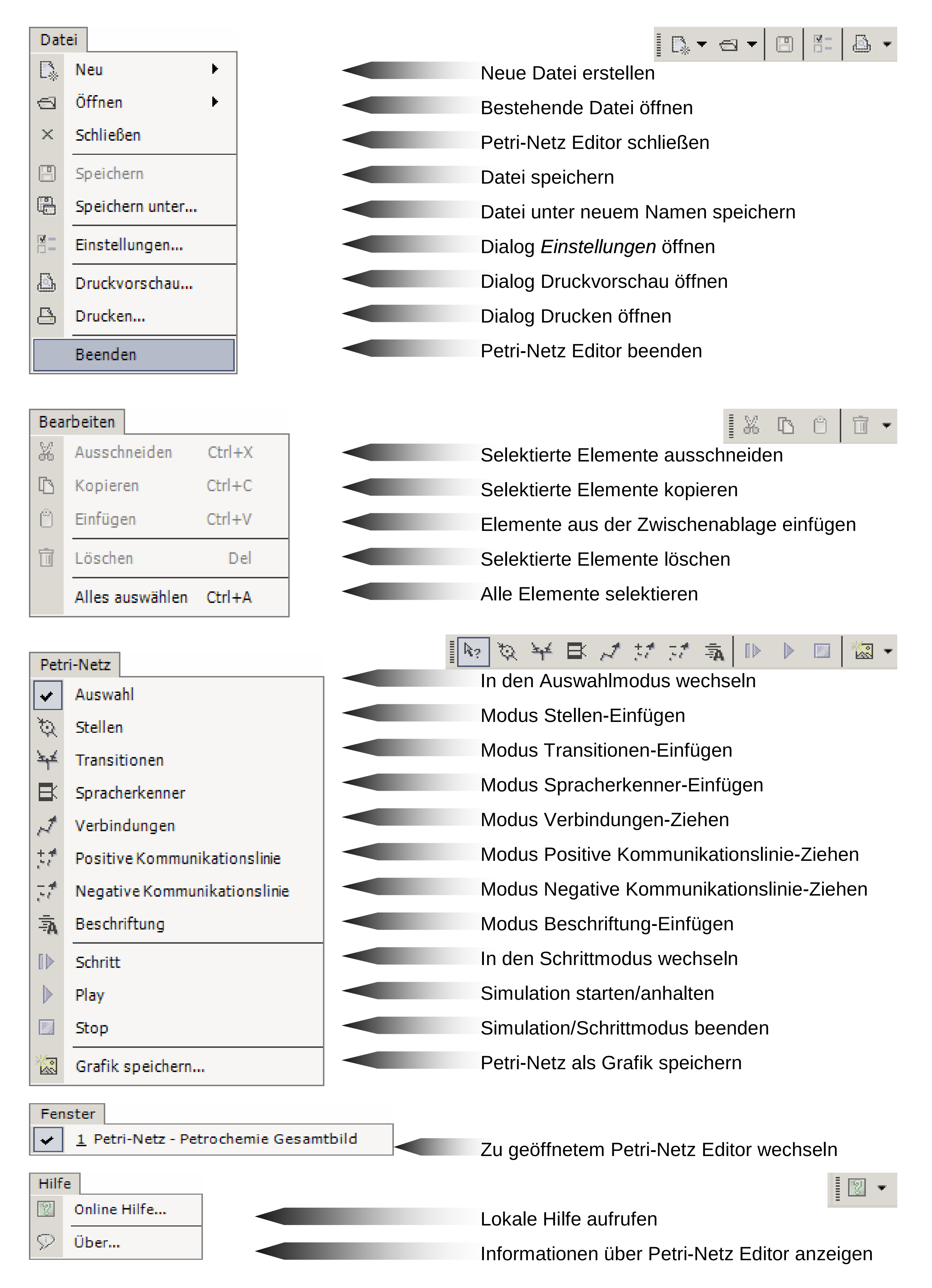
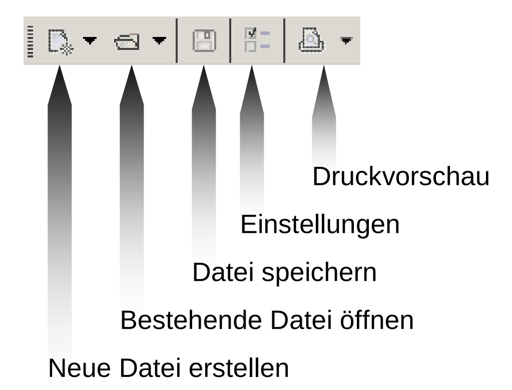
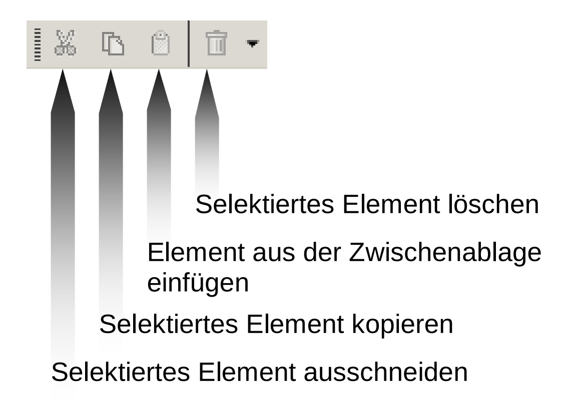
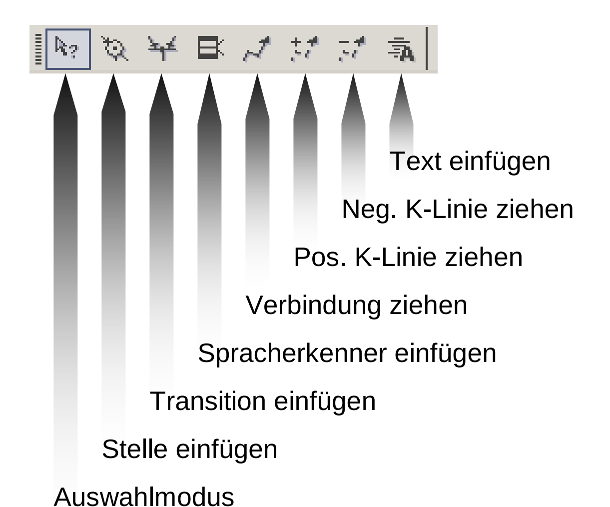


Abb. 4: Menü *Datei*, *Bearbeiten*, *Petri-Netz*, *Fenster* und *Hilfe*

2.4. Werkzeugleisten







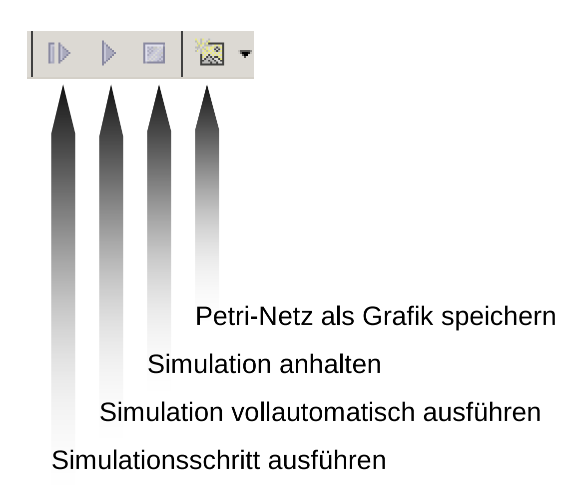


Abb. 5: Werkzeugleisten mit Datei-, Bearbeitungs-, Entwurfs- und Simulatorfunktionen

2.5 Zeichenfläche

Die Zeichenfläche stellt das aktuelle Petri–Netz dar. Im Entwurfsmodus wird ein Gitter angezeigt, an dem die Elemente und Strukturen ausgerichtet werden. Wenn die Zeichnung breiter bzw. höher als der darstellbare Bereich ist, dann werden Scrollbars eingeblendet. Die Größe der Zeichenfläche wird im Dialog *Einstellungen* eingerichtet.

2.6 Statuszeile

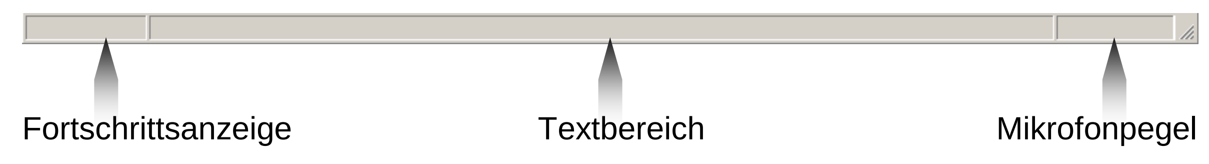


Abb. 6: Statuszeile

2.7. Einstellungen

Einstellungen

Petri-Netz
Bezeichnung Petrochemie Gesamtbild
Max. Marken pro Stelle 999
Nach Hinzufügen einer Stelle, Transition etc. in den Auswahlmodus wechseln

Sidebar
Komponentenbaum anzeigen

Seiten
Horizontal 2
Vertikal 1
Seite einrichten...
Eckjustierungen drucken

Simulation
Selektion in den Simulationsmodi verbergen
Transition-Trace loggen
Simulation animieren
Animationsintervall 250 ms (3x pro Schritt + Overhead)

Spacheingabe
Spracheingabe aktivieren
Emulieren

Spachausgabe
Sprachausgabe aktivieren
Emulieren

Abbrechen Ok

Abb. 7: Einstellungen

Petri-Netz

Geben sie hier ihrem Petri-Netz eine *Bezeichnung*. Diese Bezeichnung wird im Fenstertitel angezeigt und darf sich vom Dateinamen unterscheiden.
Die *Maximale Markenzahl pro Stelle* wird während der Simulation überprüft. Überscheitet die Markenzahl einer Stelle diese Grenze, so wird die Simulation beendet.

Seiten

Hier wird die Größe der Zeichenfläche festgelegt. Voreingestellt sind die Maße von einem A4-Blatt im Hochformat.

Simulation

Markieren sie hier die Auswahl *Transition-Trace loggen*, wenn sie im Simualtions-modus die gefeuerten Transitionen aufzeichnen möchten.
Markieren sie *Simulation animieren*, um feuernde Transitionen rot darzustellen.

2.8. Druckvorschau und Drucken

Abb. 8: Druckvorschau

In der Druckvorschau können die im Dialog Einstellungen eingerichteten Seiten betrachtet werden.
Die Seiten können entweder an den Drucker gesendet oder als PDF-Datei gespeichert werden.

Um das Petri-Netz als Grafik-Datei zu speichern, verwenden sie bitte den Befehl *Grafik speichern* aus dem Menü Petri-Netz.

2.9. Lampen

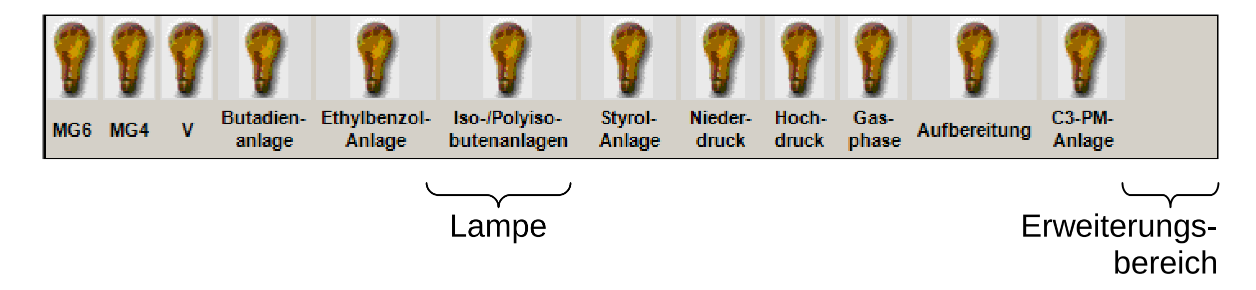


Abb. 9: Lampenanzeige

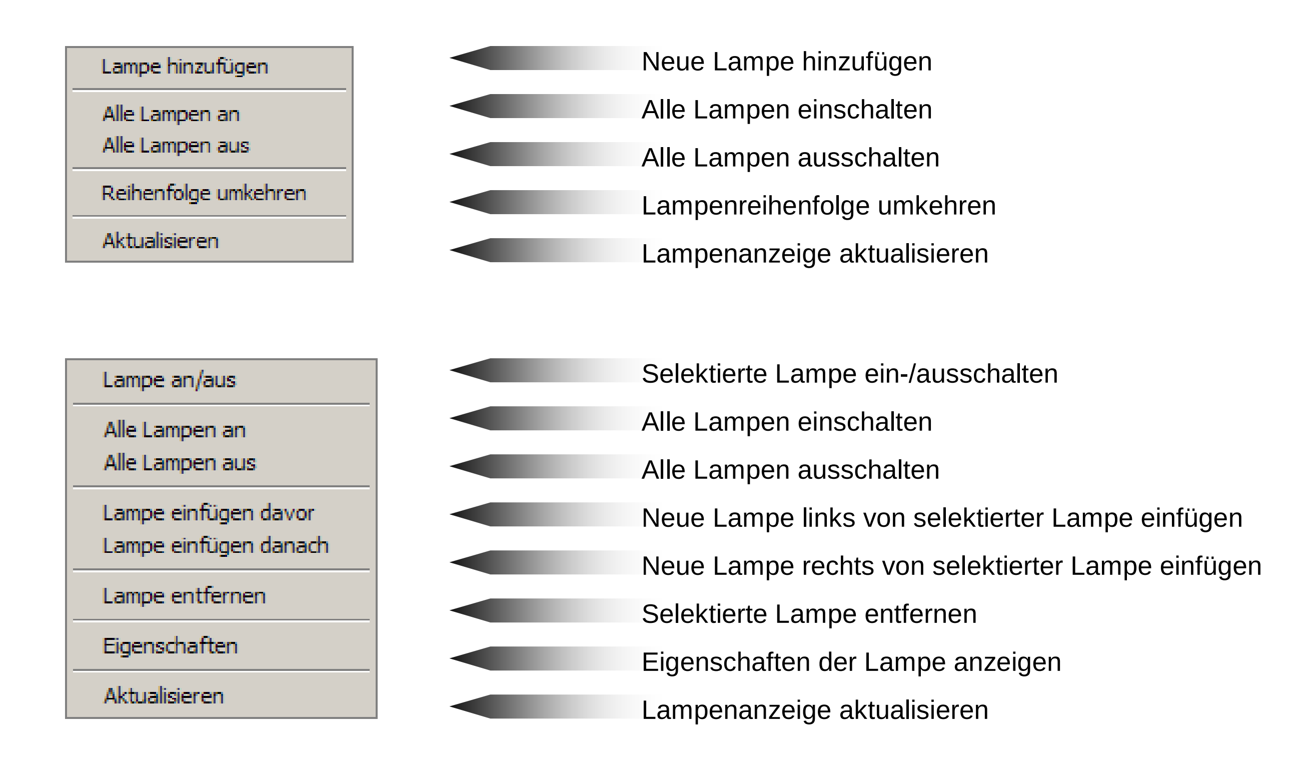


Abb. 10: Kontextmenü: Erweiterungsbereich und Lampe

Lampeneinstellungen
Allgemein
Text Ethylbenzol-Anlage
Zustand: An Aus
Zustand An

Element ID	Beschreibung
T10	Ethylbenzol Anlage

Element Hinzufügen Löschen

Liste aller Trigger-Elemente, welche die Lampe **einschalten**

Zustand Aus

Element ID	Beschreibung
T10	Ethylbenzol Anlage

Element Hinzufügen Löschen

Liste aller Trigger-Elemente, welche die Lampe **ausschalten**

Abbrechen Ok

Abb. 11: Lampeneinstellungen

3. Entwurfsmodus

In diesem Modus können Petri-Netze entworfen und verändert werden.

3.1. Auswahl-/Selektionsmodus

Im Selektionsmodus werden Elemente bzw. Strukturen der Zeichenfläche ausgewählt.
Selektierte Elemente können

- durch einfaches Anklicken und Ziehen auf der Zeichenfläche verschoben
- durch *Ausschneiden* oder *Kopieren* in die Zwischenablage transferiert
- durch *Löschen* entfernt

werden.

3.2. Stelle einfügen

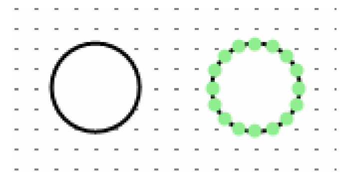


Abb. 12: Ansichten einer Stelle - normal und selektiert

Wenn der *Stellen*-Modus aktiviert ist, wird durch einen Klick auf die Zeichenfläche eine Stelle an dieser Position eingefügt. Eine Stelle wird durch einen Kreis dargestellt, in welchem die enthaltenen Marken angezeigt werden. Bis zu vier Marken werden als Punkte gezeichnet, ab 5 Marken wird die Anzahl numerisch angezeigt. Ist eine Stelle hinsichtlich ihrer Fähigkeit Marken aufzunehmen beschränkt, so wird dieser Umstand durch eine Linie aus der Stelle heraus dargestellt und deren maximale Kapazität am äußeren Ende notiert.

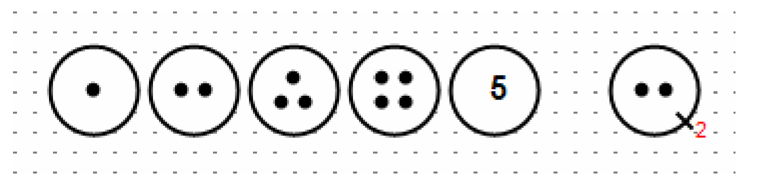


Abb. 13: Darstellungen einer Stelle (Markenzahl, Beschränkung)

Die Anzahl der enthaltenen Marken bzw. die maximale Kapazität einer Stelle können über deren Eigenschaften-Dialog geändert werden. Dieser Dialog kann entweder über einen Doppelklick auf die gewünschte Stelle oder über das Kontextmenü der Stelle aufgerufen werden. Eine Kapazität von 0 bedeutet, dass die Stelle nicht beschränkt ist (Standardeinstellung).

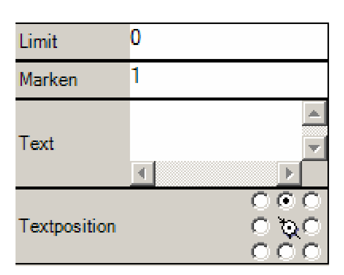


Abb. 14: *Eigenschaften* einer Stelle

Stellen werden durch Pfade mit Transitionen verbunden. Ist ein Stellenelement selektiert, so zeichnen farbige Punkte jene Orte aus, an denen Pfade mit dem Stellen-Element verbunden werden können.

3.3. Transition einfügen

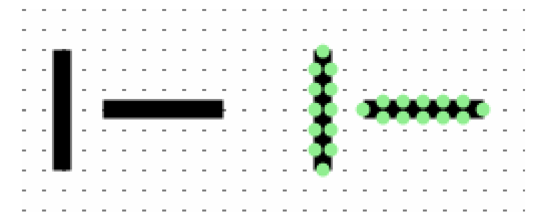


Abb. 15: Ansichten einer Transition – normal und selektiert, je horizontal und vertikal

Wenn der *Transitionen*-Modus aktiviert ist, wird durch einen Klick auf die Zeichenfläche eine Transition an dieser Position eingefügt. Diese Transition besteht aus einer dicken Linie. Die Transition kann horizontal bzw. vertikal ausgerichtet werden. Durch einen Doppelklick auf das Element kann zwischen den beiden Ausrichtungen umgeschaltet werden.

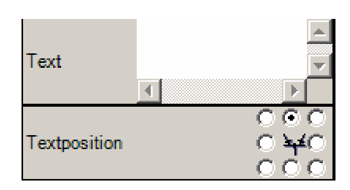


Abb. 16: *Eigenschaften* einer Transition

Transitionen werden durch Pfade mit Stellen verbunden. Ist eine Transition selektiert, so zeichnen farbige Punkte jene Orte aus, an denen Pfade mit dem Transitions-Element verbunden werden können.

3.4. Spracherkenner einfügen

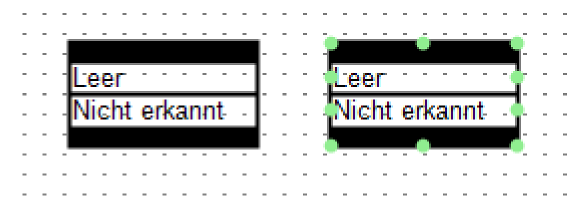


Abb. 17: Ansichten eines Spracherkenners – normal und selektiert

Im *Spracherkenner*-Modus wird durch einen Klick auf die Zeichenfläche ein Spracherkenner an dieser Position eingefügt. Dieses Element besteht aus zwei dicken Linien, welche oben und unten die Befehlsslots abgrenzen. Durch einen Doppelklick auf das Spracherkenner-Element wird dessen Dialog für die Sprachbefehle aufgerufen, über den neue Sprachbefehlslots hinzugefügt oder bestehende *Slots* gelöscht werden können.
Ein Spracherkenner ist vom Typ her eine Transition – d.h., er wird über Pfade mit Stellen verbunden.
Eingangsseitig gibt es hinsichtlich der Selektionspunkte keine Unterscheidungen. Ein Spracherkenner ist aktiviert, wenn an seiner Stelle eine Transition aktiviert wäre.
Ausgangsseitig müssen die Selektionspunkte unterschieden werden.

Dabei gehören die obersten und die untersten drei Selektionspunkte zu jenem Teil des Spracherkenners, welcher eine *Transition „emuliert“* – d.h., Verbindungen, welche diesen sechs Selektionspunkten entspringen, werden beim Feuern immer berücksichtigt.
Jeder *Slot* besitzt zwei Selektionspunkte – einen Linken und einen Rechten. Sind im Zuge eines Spracherkennungsvorganges die Bedingungen zum Feuern eines Slots erfüllt, so werden jene Verbindungen, die in diesen beiden Selektionspunkten entspringen, berücksichtigt.

Ein Spracherkenner besitzt immer die beiden Slots *Leer* und *Nicht erkannt.*
Der Slot *Leer* wird dann gefeuert, wenn nach Ablauf des Erkennungstimeouts der Erkennungsbuffer leer ist.
Der Slot *Nicht erkannt* wird dann gefeuert, wenn nach Ablauf des Erkennungstimeouts der Erkennungspuffer nicht leer ist, und kein anderer Befehlsslot feuern konnte.

Hinweis: Es können auch mehrere Befehlsslots gleichzeitig feuern - sofern ihre Feuerbedingungen erfüllt sind.

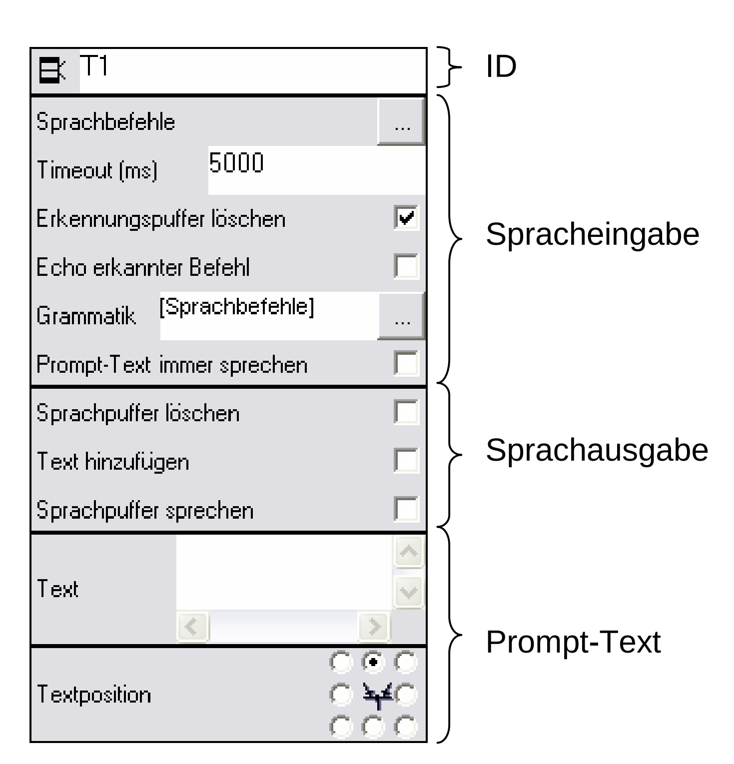


Abb. 18: *Eigenschaften* eines Spracherkenners

Sofern die drei Sprachausgabefelder *Sprachpuffer löschen*, *Text hinzufügen* und *Sprachpuffer sprechen* ausgewählt sind, wird der Inhalt des Feldes Text als Prompt-Text verwendet. Ist einleitend ein Befehlsslot bereits feuerbereit, so wird dieser Promt-Text ignoriert – es sei denn die Option *Prompt-Text immer sprechen* ist aktiviert.
Ist die Option *Erkennungspuffer löschen* markiert, so wird der Erkennungspuffer gelöscht.
Je nachdem ob das Feld *Grammatik* auf eine externe Grammatikdatei verweist oder nicht, wird nun entweder die externe Grammatikdatei eingelesen und aktiviert, oder aber eine einfache Grammatik aus den Befehlsslots generiert, welche die einzelnen Befehle als Alternativen enthält.

Sodann wird das Erkennungstimeout gesetzt und auf die Spracheingabe gewartet.
Der Spracherkenner überprüft nun in kleinen Zeitintervallen, ob Befehle erkannt wurden. Sobald Befehlsslots feuern können, wird die Spracherkennung beendet und über die entsprechenden Verbindungen gefeuert.
Ist die Option *Echo erkannter Befehl* markiert, so wird der erkannte Befehl ausgesprochen.
Abschließend kommen die Eigenschaften des/der gefeuerten Befehlsslots zum Tragen.

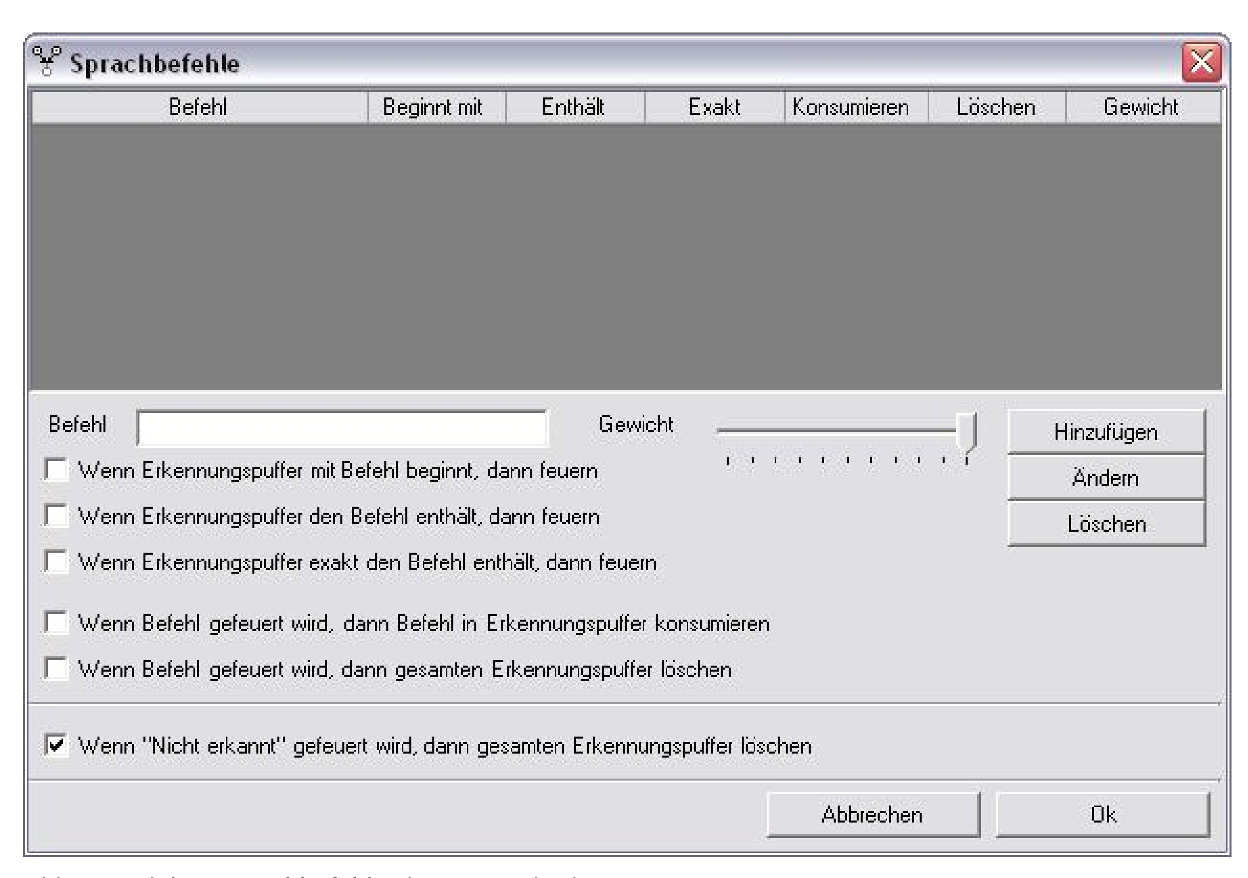


Abb. 19: Dialog *Sprachbefehle* eines Spracherkenners

3.5. Verbindung ziehen

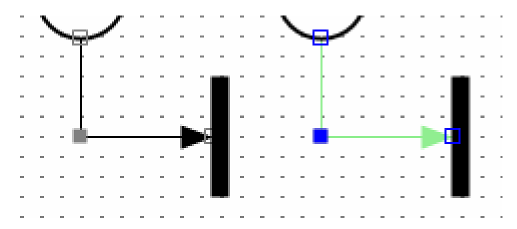


Abb. 20: Ansichten einer Verbindung – normal und selektiert

Wenn der *Verbindungs*-Modus aktiviert ist und entweder auf ein Stellen-, Transitions- oder Spracherkenner-Element geklickt wird, so wird von diesem Element ausgehend ein Pfad gezeichnet. Stützpunkte können in den Pfad aufgenommen werden, indem auf einen leeren Bereich in der Zeichenfläche geklickt wird. Wird auf ein Element geklickt, welches vom Ausgangs-Element verschieden ist, so wird dieses Element als Ziel-Element angesehen und an diesem Element der Endpunkt gesetzt. Es ist zu

beachten, dass Pfade niemals zwischen zwei gleichen Elementen gezogen werden können.

Ein Pfad beginnt bzw. endet stets an einem Selektionspunkt. Dabei wird der nächstgelegene Selektionspunkt ausgewählt. An einem Selektionspunkt können auch mehrere Pfade ihren Start-/Endpunkt haben. Durch Anklicken eines bereits selektierten Start-/Endpunktes kann der Selektionspunkt nachträglich gewählt werden. Es kann aber auch zu einem anderen Element hin verbunden werden.

Der Verlauf einer Verbindung wird über Stützpunkte gesteuert. Diese werden entweder unmittelbar während des erstmaligen Zeichnens gesetzt oder nachträglich dem Pfad über das Kontextmenü hinzugefügt. Die Stützpunkte lassen sich ebenfalls durch Anklicken und Ziehen auf der Zeichenfläche verschieben. Ein Verbindungspunkt wird gelöscht, indem man ihn zunächst selektiert und danach *Löschen* im Menü *Bearbeiten* wählt.

Handelt es sich bei einem Pfad um eine Kommunikationslinie, so kann diese sowohl positiv als auch negativ sein. Eine Kommunikationslinie wird strichliert gezeichnet und mit dem entsprechenden Vorzeichen versehen.

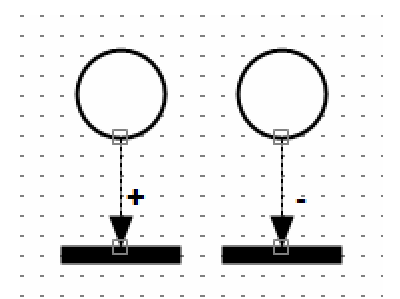


Abb. 21: Positive und negative Kommunikationslinie

Eine weitere Eigenschaft eines Pfads ist sein Kantengewicht. Wenn das Kantengewicht einer Linie größer als 1 ist, so wird die Linie dicker gezeichnet. Der Petri-Netz Editor zieht Verbindungen und Kommunikationslinien automatisch zusammen, wenn diese in gleicher Richtung die gleichen Elemente verbinden. Es wird somit kein neuer Pfad gezeichnet, sondern lediglich das Kantengewicht des vorhandenen Pfads um eins erhöht.

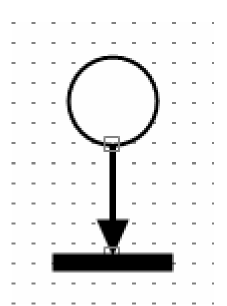


Abb. 22: Pfad mit einem Kantengewicht > 1

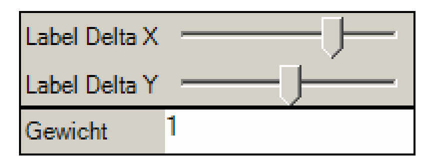


Abb. 23: *Eigenschaften* einer Kommunikationslinie

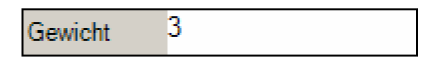


Abb. 24: *Eigenschaften* einer Verbindung

3.6. Beschriftung einfügen

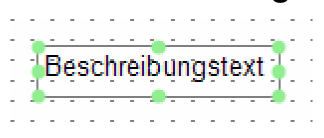


Abb. 25: Beispiel für eine Beschriftung

Elemente und Strukturen können mit Texten gekennzeichnet, bzw. der Ablauf beschrieben werden. Ein Textbaustein kann beliebig auf der Zeichenfläche positioniert werden. Durch einen Doppelklick auf einen bereits selektierten Textbaustein wird dieser in den Editiermodus gebracht. Im Eigenschafts-Bereich kann der Text jederzeit editiert werden.



Abb. 26: *Eigenschaften* einer Beschriftung

3.7. Lampen hinzufügen und verknüpfen

Um die Lampen anzuzeigen, wählt man zuerst im Kontextmenü *Anzeige* → *Zeige Lampen*. Danach im Kontextmenü der Lampenanzeige *Lampe hinzufügen*. Es wird nun eine neue Lampe der Lampenanzeige hinzugefügt.

Durch einen Doppelklick auf die neue Lampe wird ihr Dialogfenster für die Lampeneinstellungen geöffnet. Siehe auch Punkt 2.9. Lampen.
Fügen sie der Liste „*Zustand An*“ jene Elemente hinzu, welche beim Feuern die Lampe einschalten sollen.
Fügen sie entsprechend der Liste „*Zustand Aus*“ jene Elemente hinzu, welche beim Feuern die Lampe ausschalten sollen.

4. Simulationsmodus



Abb. 27: Werkzeugleiste mit Simulatorfunktionen

Der Simulationsmodus dient dazu Petri-Netze auszuführen. Die Simulation kann entweder schrittweise – auf eine interaktive Art und Weise – oder vollautomatisch erfolgen. Dabei kann zwischen beiden Ausführungsmodi gewechselt werden.

Wird eine Simulation vollautomatisch durchgeführt, so sind keine Änderungen am Petri-Netz möglich. Bei einer schrittweisen Ausführung existieren hingegen zweierlei Eingriffsmöglichkeiten:

- <u>Marken ändern:</u> durch Doppelklicken auf eine Stelle wird diese in den Editiermodus gewechselt, wo sodann die Markenzahl verändert werden kann.
- <u>Transitionen feuern:</u> sofern eine Transition feuerbereit ist, kann diese durch einfaches Anklicken zum Feuern gebracht werden.

Im Simulationsmodus sind folgende zwei Optionen von Bedeutung:

- <u>Animationsintervall:</u> gibt die Verzögerung zwischen zwei aufeinander folgenden Ausführungsschritten in Millisekunden an. Der Parameter ist nur bei der vollautomatischen Ausführung relevant.
- <u>Transition-Trace loggen:</u> fügt die gefeuerten Transitionen einer Trace-Liste hinzu. Im Schrittmodus kann ein Rollback durchgeführt werden.

Generell sind feuerbereite Transitionen schwarz dargestellt, nicht feuerbereite Transitionen sind grau. Sind vor einem Ausführungsschritt mehrere Transitionen feuerbereit, so wird zufällig eine Transition ausgewählt. Dabei ist die Auswahlwahrscheinlichkeit für jede feuerbereite Transition gleich hoch.

4.1. Schrittweise Ausführung



Abb. 28: Knopf für die schrittweise Ausführung

Wird dieser Knopf gedrückt, so wechselt der Petri-Netz Editor in den Simulationsmodus und führt das Petri-Netz schrittweise aus. Ein einmaliges Drücken dieses Knopfs führt genau einen Simulationsschritt durch.
Es können entweder weitere Einzelschritte ausgeführt werden, oder die vollautomatische Ausführung kann gestartet werden.
Durch Drücken auf den Stopp-Knopf wird die Simulation beendet und in den Entwurfsmodus gewechselt.

Wurde im Dialog Einstellungen die Option *Transition-Trace loggen* markiert, so wird bei der Abarbeitung des Petri-Netzes automatisch eine Liste der gefeuerten Transitionen erstellt und ergänzt. Der Schrittmodus erlaubt es nun, das Petri-Netz in einen vorhergehenden Zustand zurückzuführen. Hierzu muss im Kontextmenü des Transition-Trace Fensters der Menüpunkt *Rollback zu Schritt x* gewählt werden.

4.2. Vollautomatische Ausführung



Abb. 29: Knopf für die vollautomatische Ausführung

Wird dieser Knopf gedrückt, so wechselt der Petri-Netz Editor in den Simulationsmodus und führt das Petri-Netz vollautomatisch aus. Wurde das Petri-Netz vorher im Schrittmodus ausgeführt, so wird an der aktuellen Stelle fortgesetzt. Die Ausführung erfolgt schrittweise, wobei zwischen den einzelnen Schritten eine einstellbare Pause eingelegt wird. Dieser Zustand kann durch Drücken auf den Pause-Knopf (es wird in den Einzelschritt Modus gewechselt) oder auf den Stopp-Knopf (es wird die Simulation beendet und in den Entwurfsmodus gewechselt) verlassen werden.

5. Beispiele

Nachfolgend findet man einige einfache Beispiele, die den Umgang mit dem Petri-Netz Editor näher bringen sollen.

5.1 Beispiel 1 – Minimum

Aufgabe: Implementieren sie eine binäre Minimum-Operation in einem Petri-Netz derart, dass die beiden Operanden als Markenzahl in zwei korrespondierenden Stellen abgebildet werden. Ermitteln sie das Minimum der Markenzahl dieser beiden Stellen.

Vorgehensweise: Über eine Transition werden solange gleichzeitig in beiden Operanden-Stellen Marken abgezogen und einer Ergebnis-Stelle hinzugefügt, bis einer der beiden Operanden-Stellen keine Marken mehr hat.

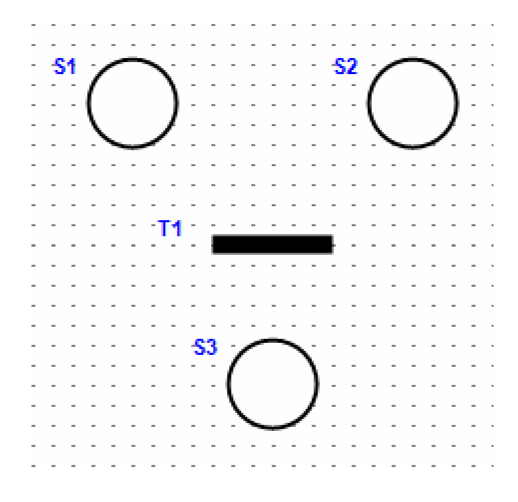
5.1.1. Entwerfen des Petri-Netzes

Wir benötigen für das Petri-Netz also drei Stellen und eine Transition.

Stellen und Transitionen einfügen

Schalten sie zuerst in den Stellen-Modus und fügen sie an einer beliebigen Position durch einen Klick auf die Zeichenfläche eine Stelle ein.
Fügen sie auf diese Weise noch zwei weitere Stellen ein. Bilden sie mit den Stellen ein auf der Spitze stehendes Dreieck.
Schalten sie nun in den Transitionen-Modus und fügen sie in der Mitte des Dreiecks durch einen Klick eine horizontale Transition ein.

Nun müssten sie folgendes Bild vor sich haben:

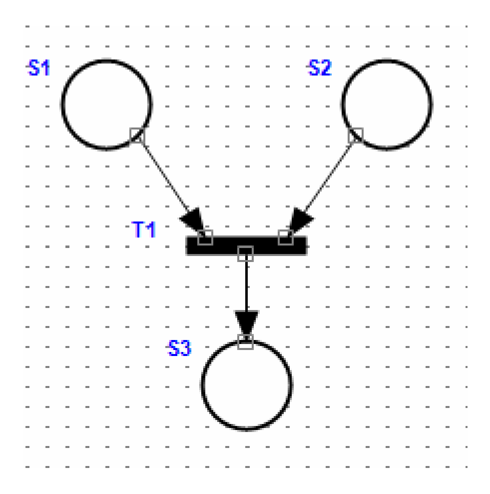


Verbindungen ziehen

Schalten sie nun in den Verbindungs-Modus und verbinden sie die Stelle S1 mit der Transition T1. Wiederholen sie dieses Verfahren und verbinden sie die Stelle S2 mit der Transition T1 und abschließend noch einmal die Transition T1 mit der Stelle S3.

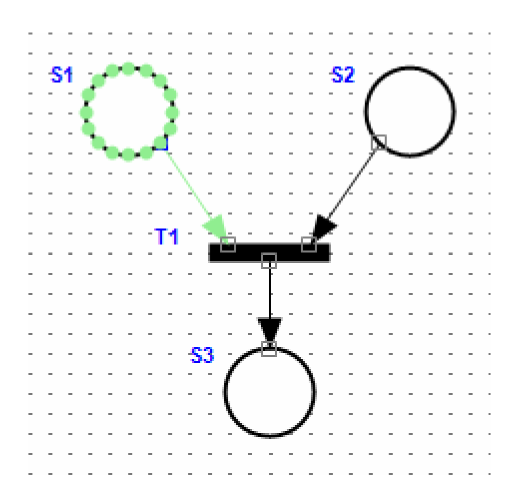
Hinweis: Sie können jederzeit den Verbindenprozeß mit der Escape-Taste abbrechen, oder nachträglich die Verbindungen verändern.

Sie sollten nun folgendes Bild vor sich haben:

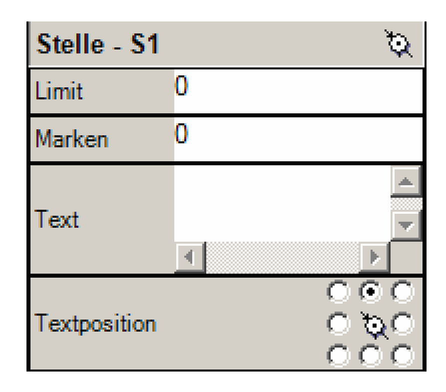


Beschriftungen

Selektieren sie nun die Stelle S1, indem sie einmal auf die Stelle S1 klicken. Achten sie darauf, dass sich der Petri-Netz Editor auch im Auswahl-/Selektionsmodus befindet.

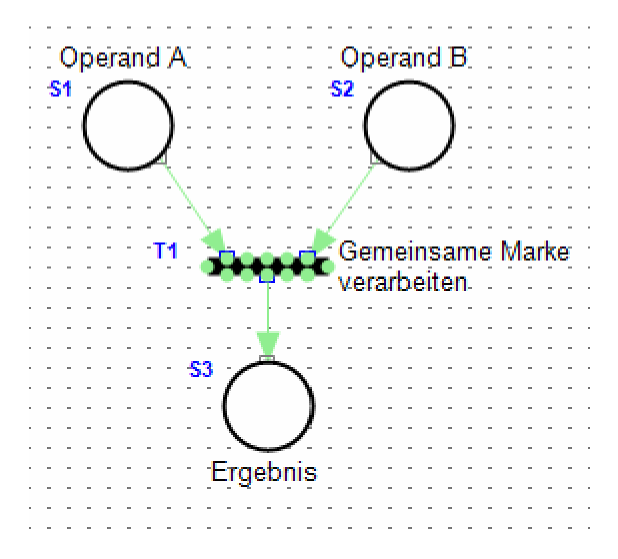


Beachten sie, dass nun in der rechten unteren Ecke des Petri-Netz Editors die Eigenschaften dieser Stelle angezeigt werden.



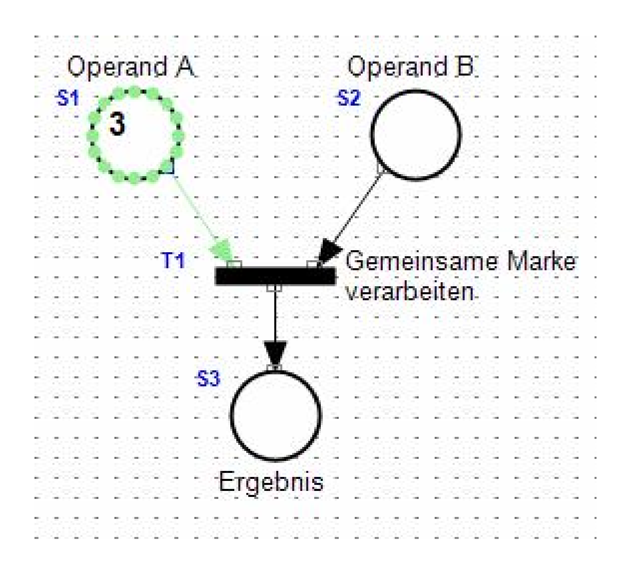
Geben sie nun im Feld *Text* den Text „Operand A“ ein.
Beschriften sie auf diese Weise die Stelle S2 mit „Operand B“, die Stelle S3 mit „Ergebnis“ und die Transition T1 mit „Gemeinsame Marke verarbeiten“.

Sie haben nun das folgende Bild vor sich:



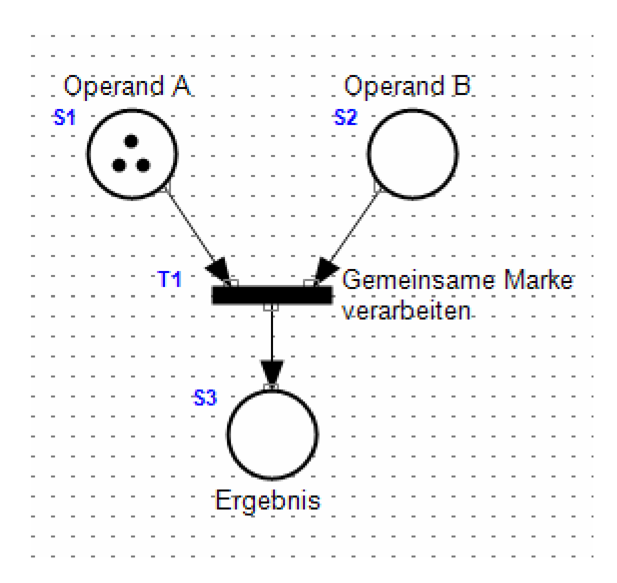
Marken setzen

Doppelklicken sie auf der Stelle S1. Die Stelle wechselt daraufhin in den Editiermodus, und sie können die Markenzahl eingeben. Geben sie in der Stelle S1 den Wert 3 ein.

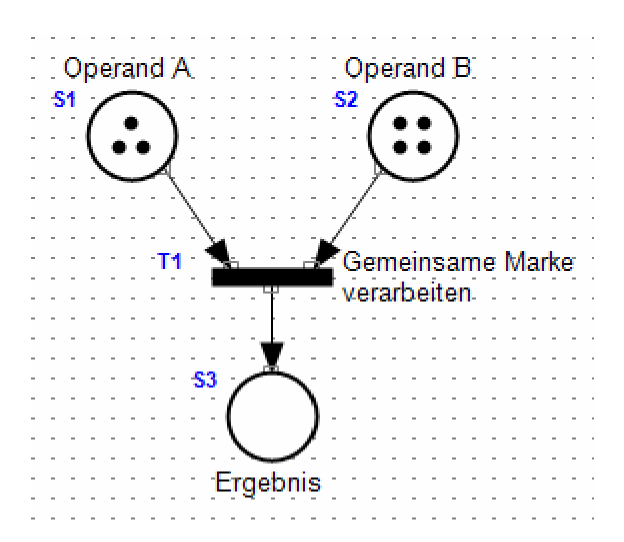


Klicken sie zum Abschluss der Eingabe auf einen leeren Zeichenbereich, um die Eingabe in die Stelle zu übernehmen.

Hinweis: Alternativ können sie den Wert natürlich auch im Eigenschafts-Bereich im Feld *Marken* abändern.



Wiederholen sie diese Vorgehensweise für die Stelle S2 und geben sie dort den Wert 4 ein.



Nun ist das Petri-Netz für den ersten Simulationslauf bereit.

5.1.2. Testen des Petri-Netzes

Stellen sie sicher, dass in den Einstellungen des Petri-Netzes die Option *Transition-Trace loggen* markiert ist, sonst bekommen sie keine Liste der gefeuerten Transitionen geliefert.

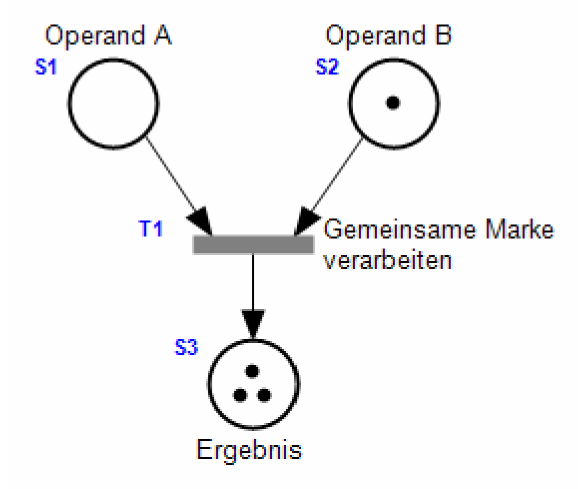
Automatische Simulation

Wechseln sie nun in den Simulationsmodus, indem sie auf den *Play*-Knopf in der Toolbar drücken, und beobachten sie was passiert.

Nach genau drei Schritten bleibt die Simulation stehen. In der Ergebnis-Stelle stehen drei Marken. Das Transition-Trace-Fenster zeigt folgende drei Schritte an:

Schritt	Transition ID	Bezeichnung
1	T1	Gemeinsame Marke verarbeiten
2	T1	Gemeinsame Marke verarbeiten
3	T1	Gemeinsame Marke verarbeiten

Das abgearbeitete Petri-Netz sieht nun folgendermaßen aus:

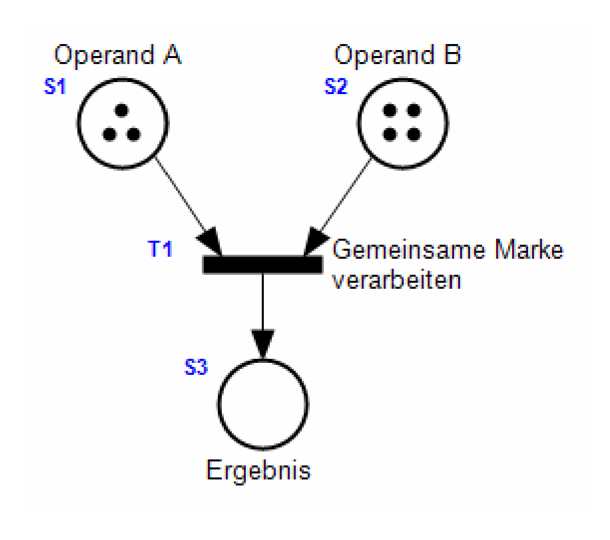


Drücken sie nun den Stopp-Knopf in der Toolbar, und wechseln sie zurück in den Auswahl-/Selektionsmodus.

Wir wollen uns nun die Simulation schrittweise ansehen.

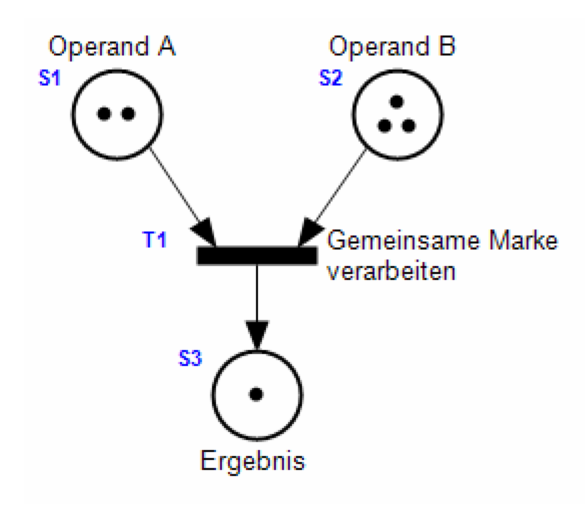
Schrittweise Simulation

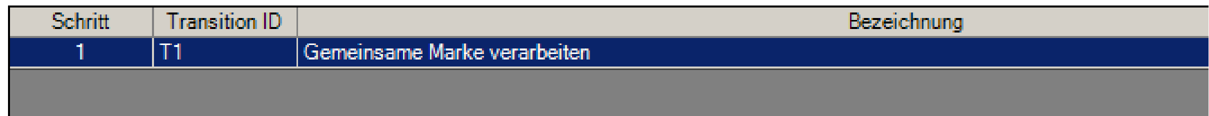
Drücken sie zunächst einmal auf den Schritt-Knopf in der Toolbar, um den Schritt-Modus zu aktivieren. Beachten sie, dass dabei der Schritt-Knopf gedrückt bleibt.



Beachten sie auch, dass die Transition T1 schwarz gefärbt, d.h., feuerbereit ist.

Klicken sie nun einmal auf die Transition T1, um sie einmal zu feuern. Wurde in den Einstellungen die Option Simulation animieren markiert, so leuchtet die Transition kurz rot auf, um ihr Feuern zu versinnbildlichen.
Gleichzeitig wird im Transition-Trace-Fenster ein Eintrag hinzugefügt.

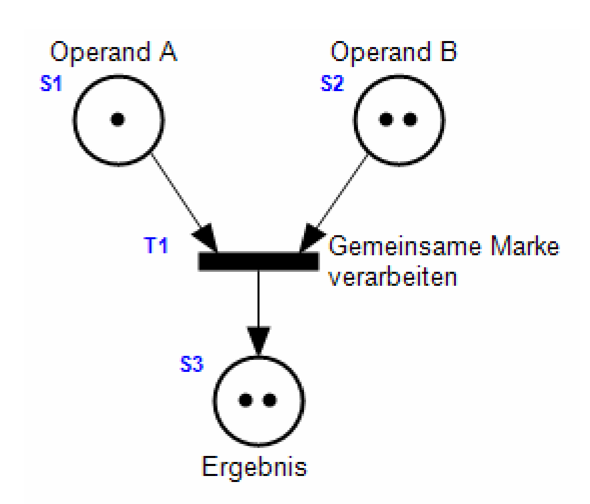




Schritt	Transition ID	Bezeichnung
1	T1	Gemeinsame Marke verarbeiten

In diesem Schritt wurde also von den Stellen S1 und S2 je eine Marke abgezogen, und eine Marke der Stelle S3 hinzugefügt.

Klicken sie nun erneut einmal auf die Transition T1 und beobachten sie, was passiert.



Schritt	Transition ID	Bezeichnung
1	T1	Gemeinsame Marke verarbeiten
2	T1	Gemeinsame Marke verarbeiten

Erneut wurden von den Stellen S1 und S2 je eine Marke abgezogen, und eine Marke der Stelle S3 hinzugefügt. Beachten sie, dass die Transition noch immer feuerbereit ist, da sie noch immer der Schaltbedingung genügt.

Ein weiterer Klick auf die Transition bringt uns ans Ende der Simulation und führt zum Ergebnis des automatischen Simulationslaufes. Die Transition ist nun grau eingefärbt und lässt sich nicht mehr weiter feuern.

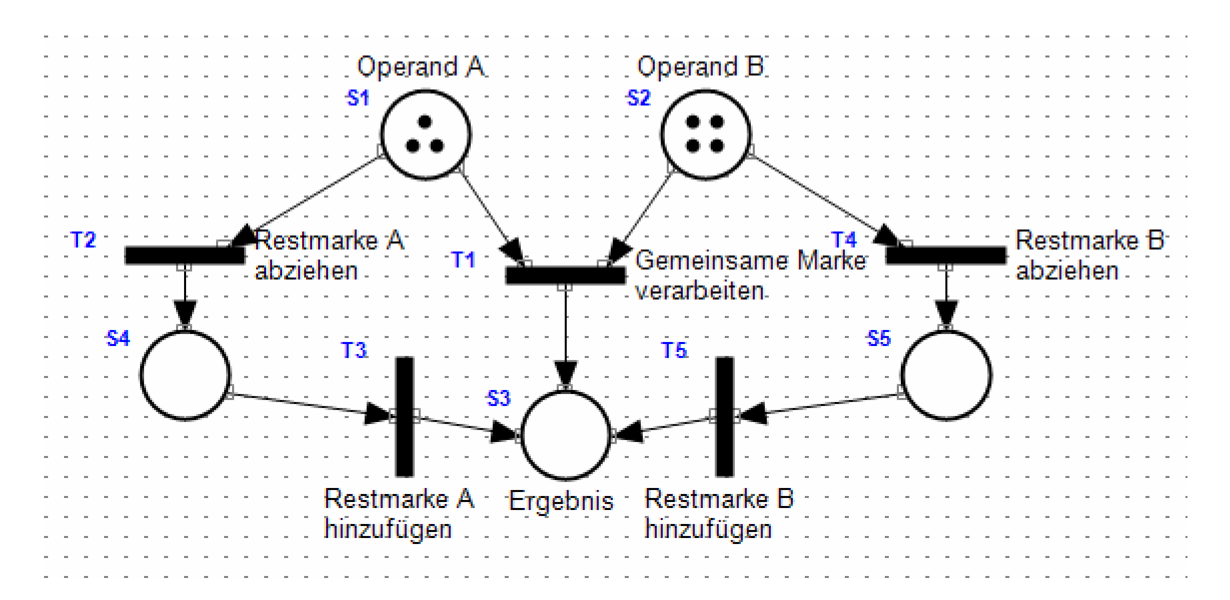
5.2 Beispiel 2 – Maximum

Aufgabe: Implementieren sie eine binäre Maximum-Operation in einem Petri-Netz derart, dass die beiden Operanden als Markenzahl in zwei korrespondierenden Stellen abgebildet werden. Ermitteln sie das Maximum der Markenzahl dieser beiden Stellen.

Vorgehensweise: Wir werden nun - aufbauend auf dem vorherigen Beispiel - eine Maximum-Funktion implementieren, in dem wir einfach den verbleibenden Rest in den Operanden-Stellen der Minimum-Funktion der Ergebnis-Stelle hinzufügen.

5.2.1. Entwerfen des Petri-Netzes

Erstellen sie anhand der Anleitung aus dem vorherigen Beispiel das folgende Petri-Netz:

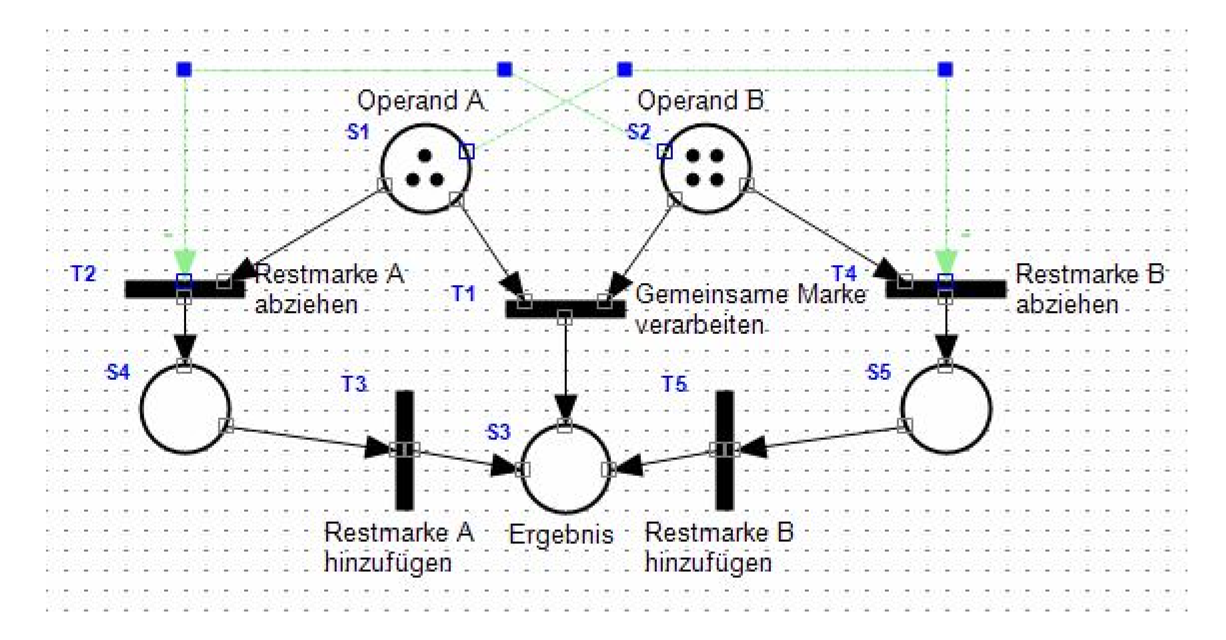


Beachten sie, dass dieses Petri-Netz noch keine Lösung des Problems darstellt, da hier als Ergebnis vom Maximum bis hin zur Addition beider Operanden alles möglich ist. Wir müssen sicherstellen, dass jene Transitionen welche Restmarken von den Operanden abziehen (T2 und T4) erst dann feuern können, wenn bereits das Minimum beider Operanden im Ergebnis steht. D.h., wenn der gegenüberliegende Operand (für T2 ist das S2 und für T4 ist das S1) keine Marken mehr enthält.

Bedingungen der Form „*Stelle muss <u>mindestens **x**</u> Marken enthalten, damit eine Transition feuern kann*“ und „*Stelle darf <u>höchstens **y-1**</u> Marken enthalten, damit eine Transition feuern kann*“ werden durch eine Positive Kommunikationslinie (mit Kantengewicht **x**) und eine Negative Kommunikationslinie (mit Kantengewicht **y**) realisiert.

In unserem Fall benötigen wir zwei Negative Kommunikationslinien mit je einem Kantengewicht von 1.

Daraus ergibt sich das folgende Petri-Netz (die beiden hinzugefügten Negativen Kommunikationslinien sind grün selektiert):

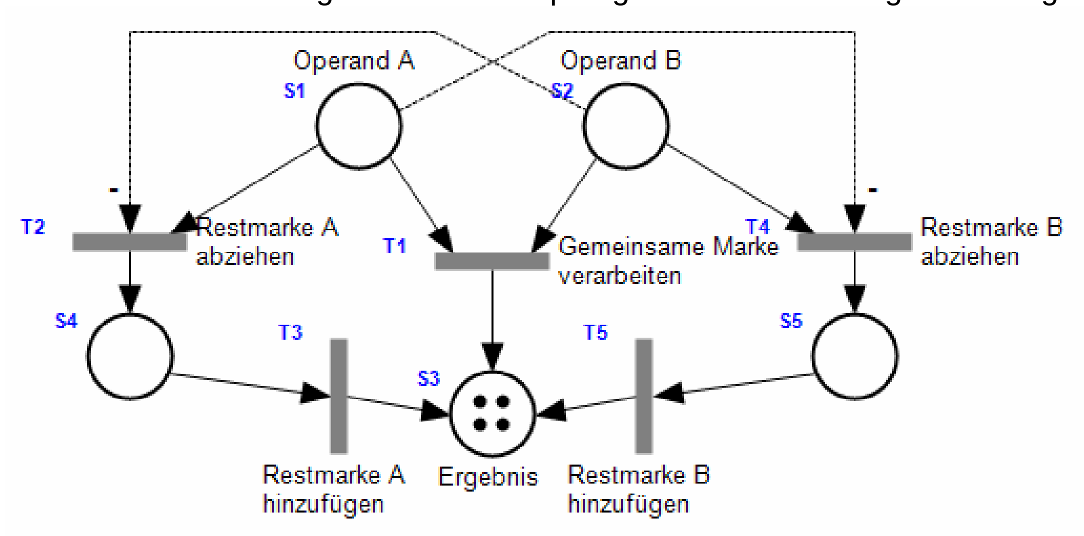


Die eine Negative Kommunikationslinie verläuft von der Stelle S1 (Operand A) hin zur Transition T4 (Restmarke B abziehen) und blockiert diese solange, bis Operand A keine Marken mehr hat.
Die andere Negative Kommunikationslinie verläuft von der Stelle S2 (Operand B) hin zur Transition T2 (Restmarke A abziehen) und blockiert diese solange, bis Operand B keine Marken mehr hat.

Insgesamt gesehen wird also zuerst das Minimum der beiden Operanden in die Ergebnis-Stelle geschrieben und danach über einen Seitenpfad der noch verbleibende Rest hinzugefügt.

5.2.2. Testen des Petri-Netzes

Ein Simulationslauf bestätigt unsere Behauptungen und führt zu folgendem Ergebnis:



Schritt	Transition ID	Bezeichnung
1	T1	Gemeinsame Marke verarbeiten
2	T1	Gemeinsame Marke verarbeiten
3	T1	Gemeinsame Marke verarbeiten
4	T4	Restmarke B abziehen
5	T5	Restmarke B hinzufügen

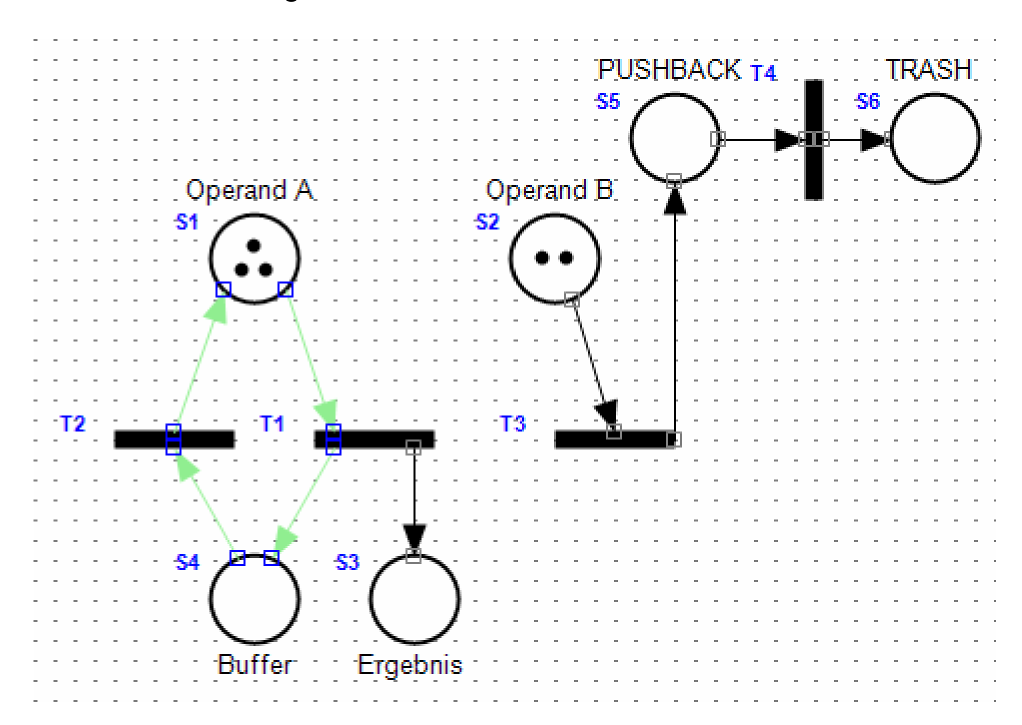
5.3 Beispiel 3 - Multiplikation

Aufgabe: Implementieren sie eine Multiplikations-Operation in einem Petri-Netz derart, dass die beiden Operanden als Markenzahl in zwei korrespondierenden Stellen abgebildet werden. Ermitteln sie das Ergebnis einer Multiplikation der Markenzahlen dieser beiden Stellen.

Vorgehensweise: Die Idee besteht darin, in einem zyklischen Verfahren die Marken von Operand A einer Ergebnis-Stelle hinzuzufügen. Jeweils am Zyklusende wird dem Operanden B eine Marke abgezogen. Das ganze Verfahren dauert solange, bis Operand B keine Marken mehr besitzt.

5.3.1. Entwerfen des Petri-Netzes

Entwerfen sie zunächst folgendes Petri-Netz:



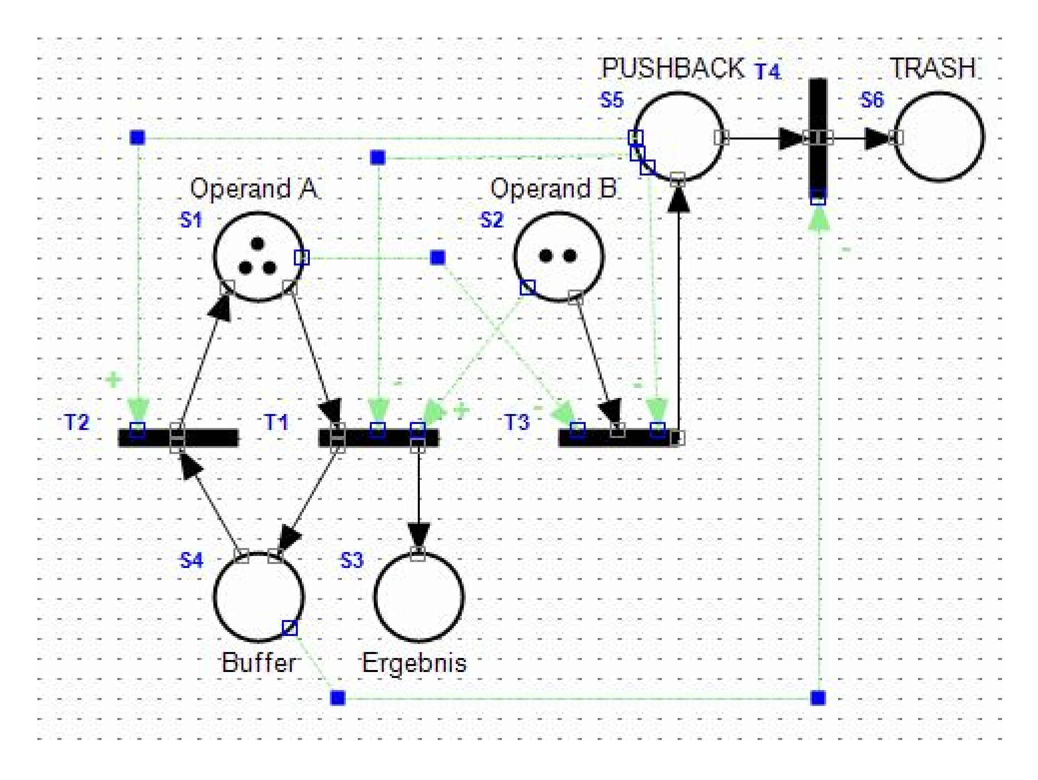
Der Zyklus mit dem die Marken von Operand A in die Ergebnis-Stelle geschrieben werden ist grün selektiert. Das rechte, nicht selektierte Teilnetz repräsentiert den Abzug der Marken aus dem Operanden B.

Folgende Situationen müssen nun sichergestellt werden:

1. T3 darf nur dann feuern,
   wenn Operand A keine Marken mehr hat (-> Neg. K-Linie, Gewicht 1),
   wenn Operand B mindestens eine Marke hat (durch S2 -> T3 erfüllt),
   wenn PUSHBACK keine Marke hat (-> Neg. K-Linie, Gewicht 1).
2. T1 darf nur feuern,
   wenn Operand A mindestens eine Marke hat (durch S1 -> T1 erfüllt),
   wenn Operand B mindestens eine Marke hat (-> Pos. K-Linie, Gewicht 1),
   wenn PUSHBACK keine Marke hat (-> Neg. K-Linie, Gewicht 1).
3. T2 darf nur dann feuern,
   wenn Puffer mindestens eine Marke hat (durch S4 -> T2 erfüllt),
   wenn PUSHBACK eine Marke hat (-> Pos. K-Linie, Gewicht 1).

4. T4 darf nur feuern,
   wenn PUSHBACK eine Marke hat (durch S5 -> T4 erfüllt),
   wenn Puffer keine Marken mehr hat (-> Neg. K-Linie, Gewicht 1).

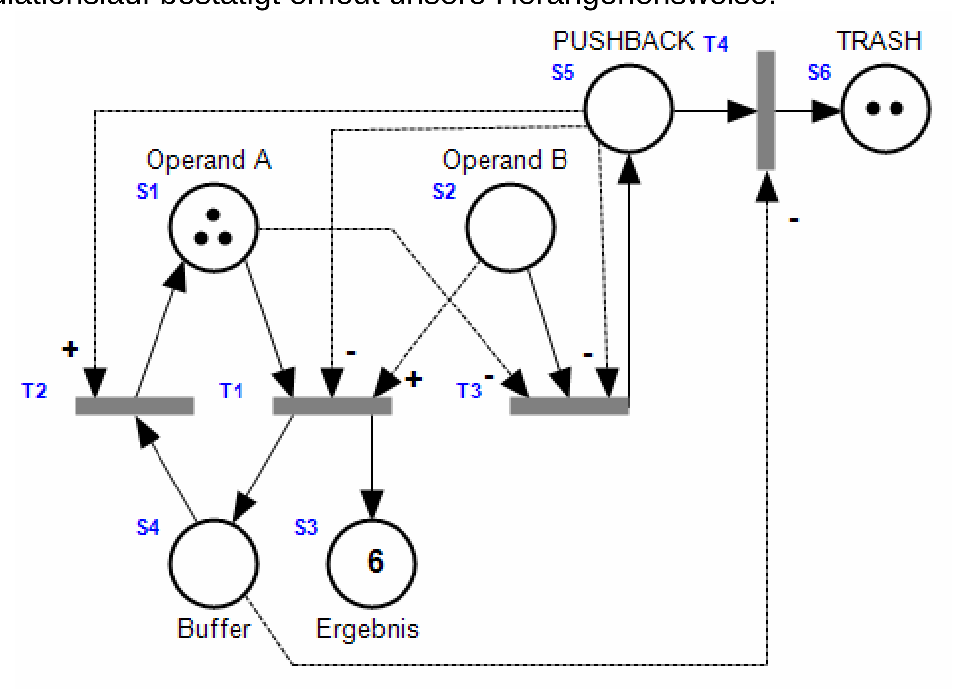
Daraus ergibt sich folgendes Petri-Netz



Sämtliche hinzugefügten Positiven und Negativen Kommunikationslinien sind grün selektiert.

5.3.2. Testen des Petri-Netzes

Ein Simulationslauf bestätigt erneut unsere Herangehensweise:



6. Anhang

6.1. Sprachausgabe

6.1.1. XML Schema: SAPI

Element	Attribute	Beschreibung
<SAPI>	-	SAPI-Tags können verschachtelt werden. Nach dem End-Tag ist der Zustand der Maschine derselbe wie vor dem Start-Tag. **Beispiele:** <SAPI>…</SAPI> <SAPI>…<SAPI>…</SAPI>…</SAPI>
<VOICE>	[REQUIRED = string] [OPTIONAL = string]	Wählt eine Stimmenimplementierung aus. **Beispiele:** <VOICE REQUIRED="NAME=Microsoft Mike">…</VOICE> <VOICE REQUIRED="NAME=Microsoft Mary">…</VOICE> <VOICE REQUIRED="NAME=Microsoft Sam">…</VOICE>
<PITCH>	[ABSMIDDLE = string] MIDDLE = string	Setzt die Stimmhöhe. **Beispiele:** <PITCH MIDDLE="10">high pitch</PITCH> <PITCH MIDDLE="-10">low pitch</PITCH>
<RATE>	[ABSSPEED = int] [SPEED = int]	Setzt die Sprachgeschwindigkeit **Beispiele:** <RATE SPEED="10">quickly spoken text</RATE> <RATE SPEED="-10">slowly spoken text</RATE>
<SILENCE>	MSEC = int	Pause für eine gewisse Zeitspanne (in Millisekunden) **Beispiele:** <SILENCE MSEC="500"/>
<SPELL>	-	Buchstabiert einen Ausdruck **Beispiele:** <SPELL>Petri-Net Editor</SPELL>
<EMPH>	-	Betont einen Ausdruck **Beispiele:** <EMPH>do</EMPH>it<EMPH>now!</EMPH>
<VOLUME>	LEVEL = int	Setzt die Lautstärke **Beispiele:** <VOLUME LEVEL="100">This is loud!</VOLUME> <VOLUME LEVEL="10">This is pretty soft!</VOLUME>

Abb. 29: Die wichtigsten Elemente und Attribute

Eine genauere Auflistung aller Elemente und Attribute findet man in der Hilfedatei zum Microsoft Speech SDK (SAPI 5.1) unter White Papers, XML Schema: SAPI [MS Speech SDK].

6.1.2. Petri-Netz-Erweiterungen für die Sprachsynthese

Vergewissern sie sich zuerst, dass sie eine SAPI-Implementierung auf ihrem Rechner installiert haben.
Im Verzeichnis **C:\Programme\Gemeinsame Dateien\ Microsoft Shared\Speech** befinden sich die Dateien **sapi.dll** sowie die Control-Panel Datei **sapi.cpl**.

Über das Control-Panel werden die Default-Einstellungen für die Sprachausgabe vorgenommen (Stimme, Lautstärke, Landessprache,…).
Wenn sie im Einstellungs-Dialog des Petri-Netzes die Option *Sprachausgabe aktivieren* markieren, erscheinen im Eigenschaften-Dialog der Transitionen drei weitere Optionen für die Sprachsynthese.

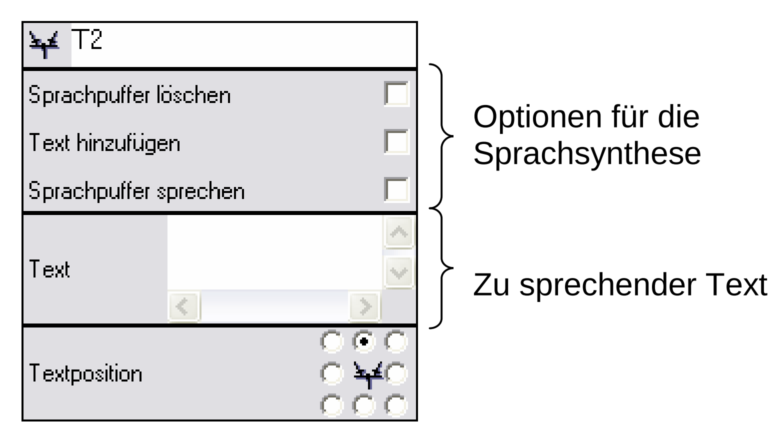


Abb. 30: Eigenschafts-Bereich einer Transition mit den Optionen für Sprachsynthese

Jeder Petri-Netz Editor besitzt einen Sprachpuffer. Feuert eine Transition, so kommen ihre Einstellungen zur Sprachausgabe zum Tragen.

Ist die Option *Sprachpuffer löschen* markiert, wird der Sprachpuffer einleitend (also vor allen anderen Sprachaktionen) gelöscht.

Ist die Option *Text hinzufügen* markiert, so wird der *Text* der Transition am Ende des Sprachpuffers angefügt.
Der *Text* darf die Tags des *XML Schemas SAPI* enthalten. Darüberhinaus existiert eine einfache **Referenzierung** der Form *$(id)*. Jede Referenzierung wird vor dem Einfügen in den Sprachpuffer durch jenen Inhalt des referenzierten Petri-Netz Elements ersetzt, der vor dem Feuern der Transition gegeben ist.

Referenziertes Petri-Netz Element	Ergebnis	Beispiel
Stelle	Markenzahl	$(S1) oder $(S$(S1))
Transition	Text	$(T1) oder $(T$(S1))
Beschriftung	Text	$(B1) oder $(B$(S1))
Verbindung	Kantengewicht	$(P1) oder $(P$(S1))
Positive Kommunikationslinie	Kantengewicht	$(P1) oder $(P$(S1))
Negative Kommunikationslinie	Kantengewicht	$(P1) oder $(P$(S1))
, Start-/Endpunkte, Stützpunkte	Kantengewicht des Pfades	$(PP1) oder $(PP$(S1))

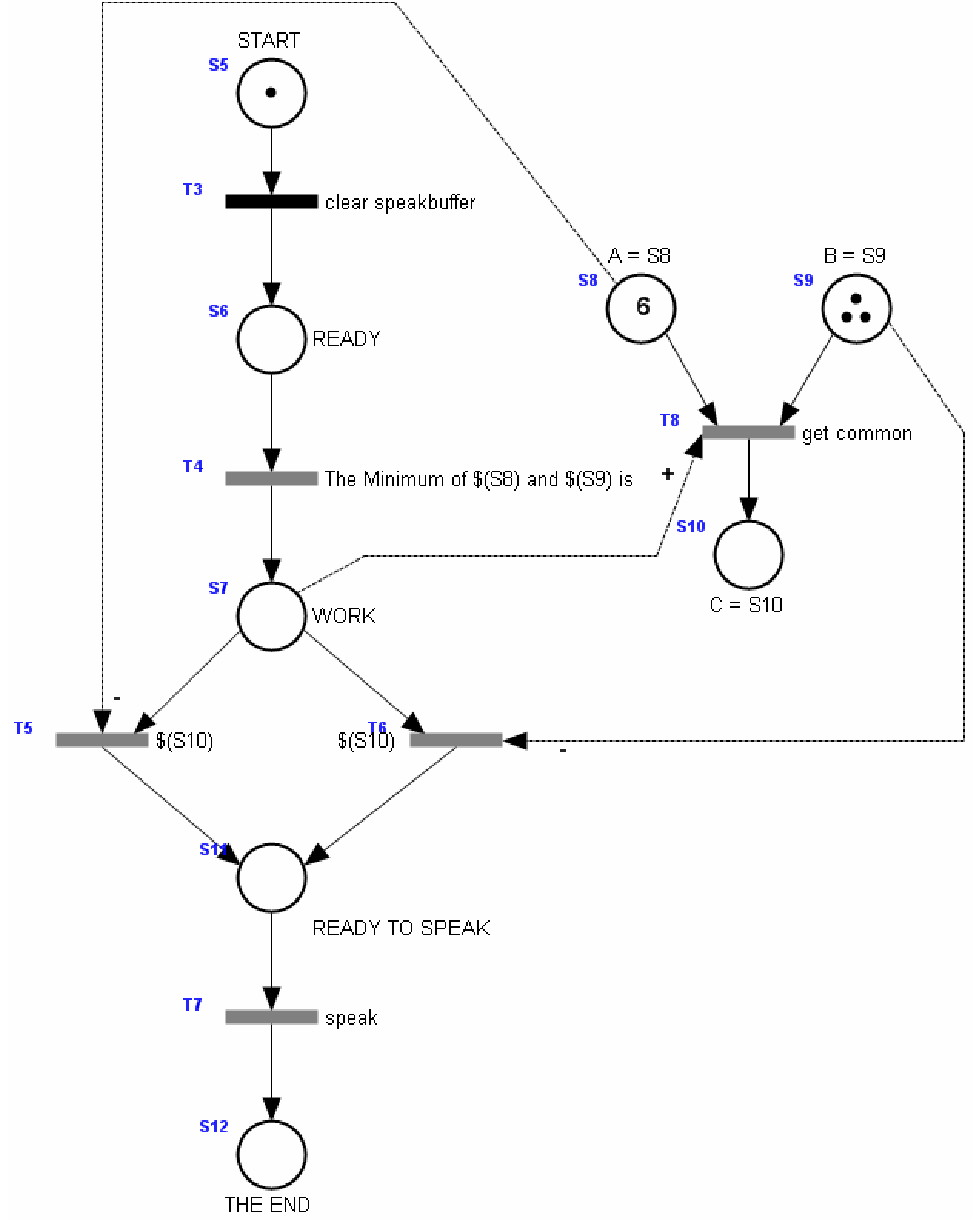
Abb. 31: Referenzierung

Wie die Beispiele zeigen, ist auch eine **Mehrfach-Referenzierung** möglich.

Ist die Option *Sprachpuffer sprechen* markiert, so wird der Sprachpuffer ausgegeben. Das Petri-Netz wartet so lange, bis der Text zu Ende gesprochen wurde und löscht abschließend den Sprachpuffer.
Sprachausgabe ist vom Typ her eine gezielte Linearisierung von Ereignissen. Es muss daher ganz besonders auf die Synchronisation der Ereignisse geachtet werden, um die Sprachausgabe nicht zu fragmentieren.

1.3. Beispiele

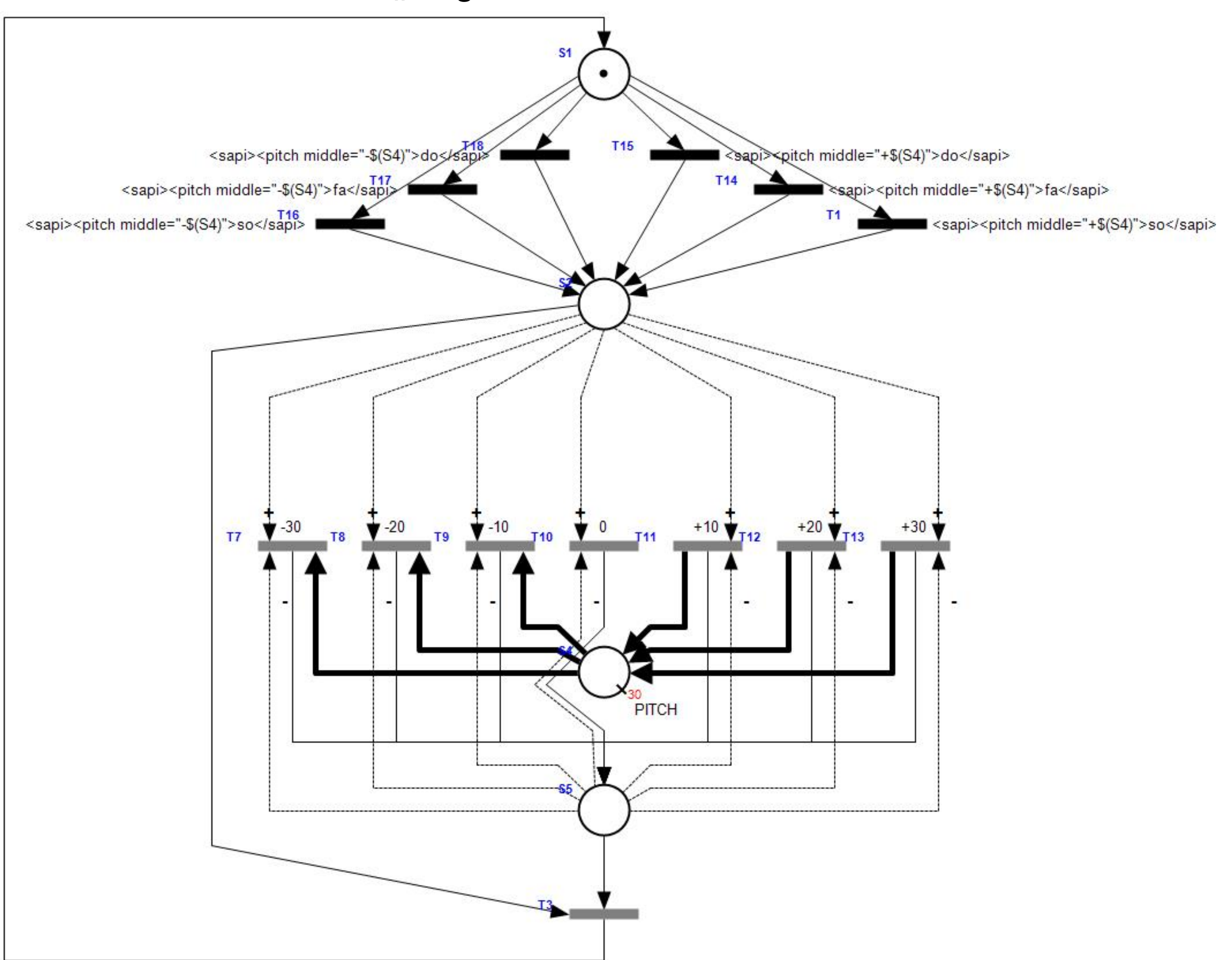
6.1.3.1. Minimum-Funktion mit Sprachausgabe



T3	…	*Sprachpuffer löschen* markiert
T4, T5, T6	…	*Text hinzufügen* markiert
T8	…	nichts markiert
T7	…	*Sprachpuffer sprechen* markiert

Sprachausgabe des Petri-Netzes: The minimum of 6 and 3 is 3.

6.1.3.2 Ein Petri-Netz lernt „singen“



Anmerkung: In den obersten 6 Transitionen sind jeweils alle drei Optionen *Sprachpuffer löschen*, *Text hinzufügen* und *Sprachpuffer sprechen* markiert. Die übrigen Transitionen unterstützen keine Sprachsynthese.

Das zentrale Element stellt die Stelle **S4** (PITCH) dar, in welchem die Stimmhöhe gespeichert ist. Ihr Wertebereich reicht von 0 bis 30.

Die unmittelbar über S4 liegenden Transitionen (T7 – T13) verändern die Markenzahl von S4 in Zehnerschritten. Die fett gezeichneten Pfade besitzen die Kantengewichte von 30,20,10,10,20 und 30 (von links nach rechts).

Die obersten 6 Transitionen interpretieren die Markenzahl der Stelle S4 als Stimmhöhe und produzieren einen der drei Ausgaben: do, fa, so

T18 … <sapi><pitch middle=“**-**$(**S4**)“>do</sapi>
T17 … <sapi><pitch middle=“**-**$(**S4**)“>fa</sapi>
T16 … <sapi><pitch middle=“**-**$(**S4**)“>so</sapi>
T15 … <sapi><pitch middle=“**+**$(**S4**)“>do</sapi>
T14 … <sapi><pitch middle=“**+**$(**S4**)“>fa</sapi>
T1 … <sapi><pitch middle=“**+**$(**S4**)“>so</sapi>

Sprachausgabe des Petri-Netzes: **[**do**,** fa**,** so**]*** in zufälligen Stimmlagen, beginnend mit der normalen Stimmlage (= 0).

6.2. Spracheingabe

6.2.1. XML Schema: Grammar

Element	Attribute	Beschreibung
<GRAMMAR>	LANGID = int	Das Top-Level XML Element, umschließt alle anderen XML Elemente, um eine Command and Control-Grammatik zu definieren **Beispiel:** <GRAMMAR LANGID="409"> … </GRAMMAR>
<RULE>	[NAME = string] [TOPLEVEL = (IN-)ACTIVE]	Ein nicht-terminierendes Regel-Element. **Beispiel:** <RULE NAME="TopLevelRule" TOPLEVEL="ACTIVE">…</RULE> **ACHTUNG!** Die oberste Regel einer Grammatik für den PED muss den <u>Namen</u> **„TopLevelRule"** haben.
<O>	-	Optionales Element: Das Element muss nicht gematched werden. **Beispiel:** <O>switch</O>on
<P>	-	Das Element muss gematched werden **Beispiel:** <P>exit</P>
<L>	-	Liste alternativer Phrasen **Beispiel:** <L> <P>one</P> <P>two</P> </L>

Abb. 32: Die wichtigsten Elemente und Attribute

Eine genauere Auflistung aller Elemente und Attribute findet man in der Hilfedatei zum Microsoft Speech SDK (SAPI 5.1) unter White Papers, XML Schema: Grammar [MS Speech SDK].

Beispiel für eine Grammatik-Datei:

```
<GRAMMAR LANGID="409">
   <RULE NAME="TopLevelRule" TOPLEVEL="ACTIVE">
      <L>
          <P>repeat</P>
          <P>exit</P>
      </L>
   </RULE>
</GRAMMAR>
```

Diese Grammatik erkennt die beiden Kommandos *repeat* und *exit*.

Beachten sie, dass der Name der obersten Regel **TopLevelRule** lauten muss, sonst wird die Grammatik im PED nicht erkannt bzw. nicht aktiviert!

6.2.2. Petri-Netz Erweiterungen für die Spracherkennung

Vergewissern sie sich zuerst, dass sie eine SAPI-Implementierung auf dem Rechner installiert haben.
Im Verzeichnis **C:\Programme\Gemeinsame Dateien\ Microsoft Shared\Speech** befinden sich die Dateien **sapi.dll** sowie die Control-Panel Datei **sapi.cpl**.
Über das Control-Panel werden die Default-Einstellungen für die Spracheingabe vorgenommen (Spracherkenner, Mikrofonempfindlichkeit,…).

Um die Spracherkennung im Petri-Netz zu nutzen, fügen sie einen Spracherkenner wie unter Punkt 3.4 beschrieben ein.

6.2.3. Beispiele

6.2.3.1 Lampensteuerung mit Spracheingabe

Aufgabe: Implementieren sie eine einfache Sprachsteuerung, mit der zwei Lampen unabhängig voneinander ein- und ausgeschaltet werden können.

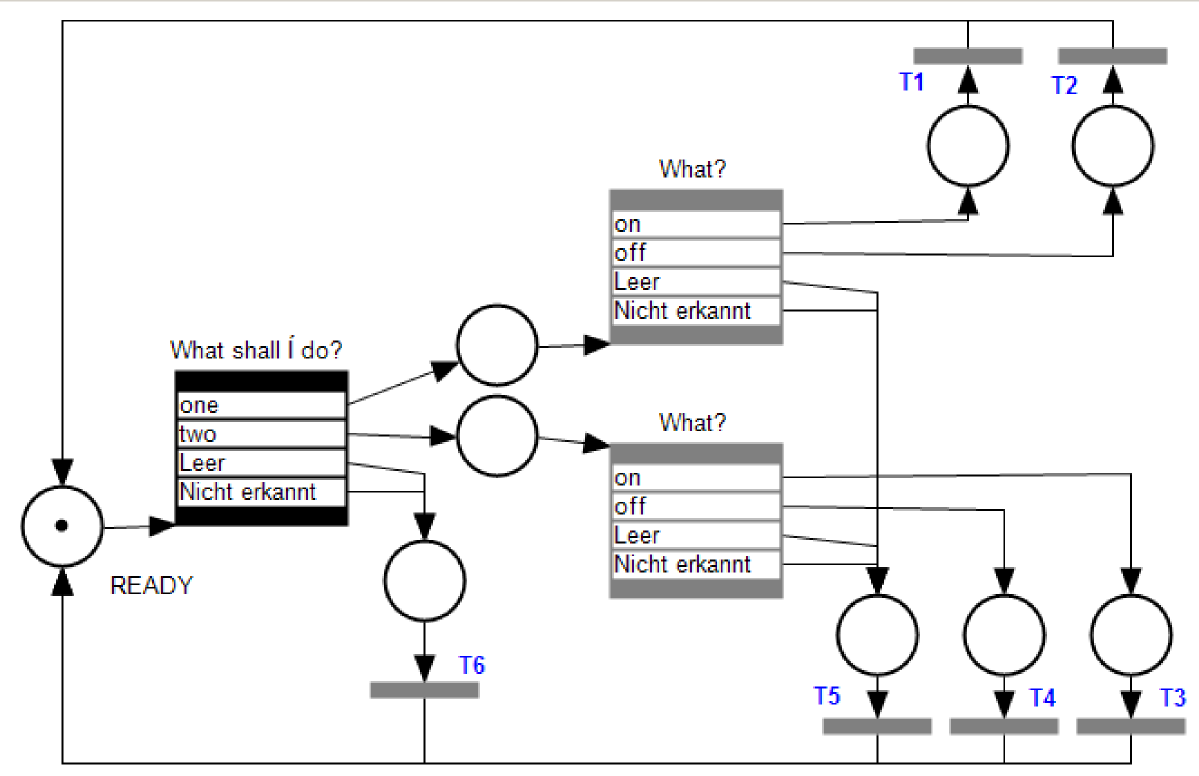
Vorgehensweise: Wir können entweder nur einen Spracherkenner verwenden und diesem vier Kommandos hinzufügen, oder aber auch drei Spracherkenner und diesen je zwei Kommandos hinzufügen. Wir wollen uns hier für die zweite Variante entscheiden und drei Spracherkenner verwenden. Der Erste soll zuerst die Lampe auswählen, der Zweite und Dritte soll festlegen, was mit dieser Lampe passieren soll. Um die Erkennung zu verbessern, verwenden wir im ersten Spracherkenner eine einfache Grammatik der Form:

[light](one,two) [switch](on,off)

<u>grammer_bsp1.xml:</u>

```
<GRAMMAR LANGID="409">
   <RULE NAME="TopLevelRule" TOPLEVEL="ACTIVE">
     <O>light</O>
     <L>
          <P>one</P>
          <P>two</P>
     </L>
     <O>switch</O>
     <L>
          <P>on</P>
          <P>off</P>
     </L>
   </RULE>
</GRAMMAR>
```





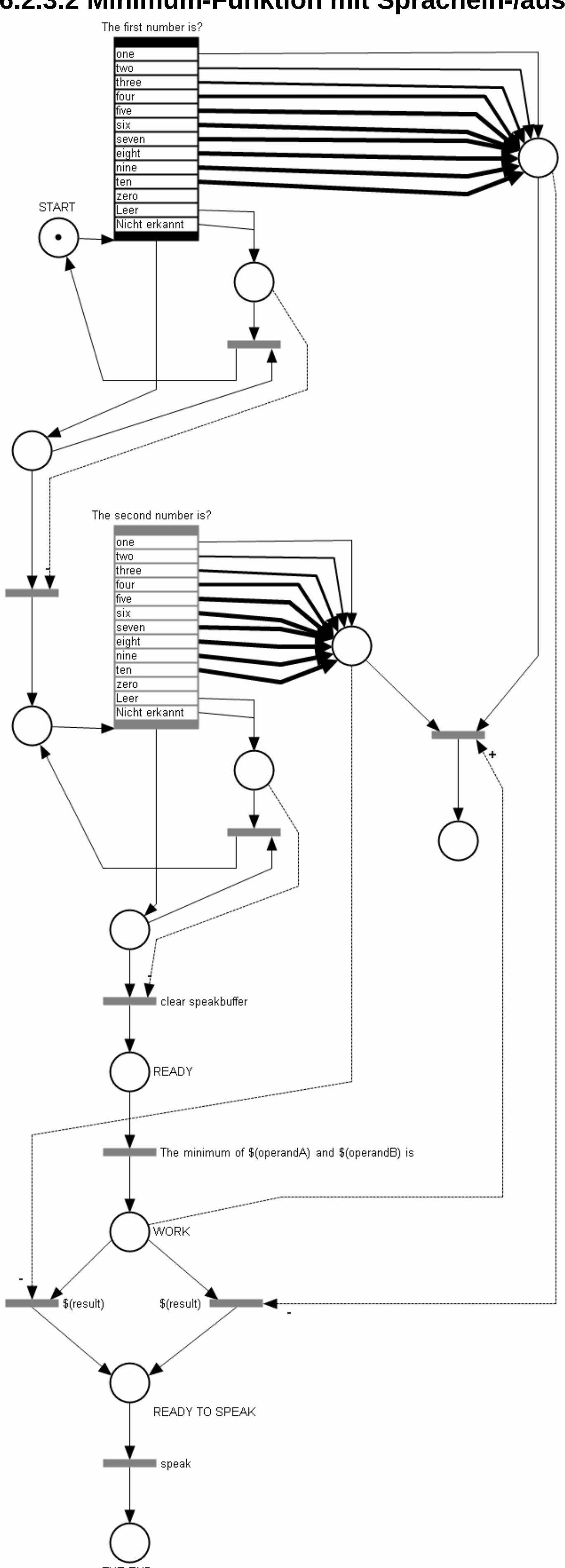
Der erste Spracherkenner stellt zuerst die Frage “what shall I do?”. Daraufhin kann z.B. „light one switch on“ als gültige Eingabe gesprochen werden. Es ist hier wichtig, dass bei den Befehlen „*one*“ und „*two*“ die Eigenschaften „*Wenn Erkennungspuffer Befehl enthält, dann feuern*“ und „*Wenn Befehl gefeuert wird, dann Befehl in Erkennungspuffer konsumieren*“ markiert sind. Im Erfolgsfall wird schließlich die entsprechende Lampe ein- oder ausgeschaltet.

In den beiden anderen Spracherkennern darf die Option „Erkennungspuffer löschen“ nicht markiert sein, da ansonsten die Information, die bereits der erste Spracherkenner erkannt hat, wieder gelöscht würde.

Die Lampe 1 enthält in der Liste „Zustand An“ die Transition T1, in der Liste „Zustand Aus“ die Transition T2.
Lampe 2 enthält in der Liste „Zustand An“ die Transition T3 und in der Liste „Zustand Aus“ die Transition T4.

Im Fehlerfall oder nach einem erfolgreichen Erkennungsvorgang beginnt der Kreislauf im Zustand READY wieder von neuem.

6.2.3.2 Minimum-Funktion mit Sprachein-/ausgabe



Aufgabe: Implementieren sie eine binäre Minimumfunktion für die Zahlen null bis zehn, welche die Zahlen abfrägt und das Ergebnis sprachlich mitteilt.

Vorgehensweise: Wir verwenden je einen Spracherkenner für jeden der beiden Operanden und schreiben die ermittelte Zahl als Markenzahl in die vorgesehene Stelle. Weitere Vorgehensweise siehe Beispiel 5.1.

<u>Eingabe:</u>
z.B. „three“, „five“

<u>Ausgabe:</u>
“The minimum of three and five is three.”

6.3. Petri-Netz API

Dem Programm liegt im Verzeichnis *lib* die Bibliothek **PetriNet.dll** bei. Diese Bibliothek kann in ein beliebiges dotNET-Projekt eingebunden werden und bietet die Möglichkeit, ein Petri-Netz, das zuvor mit dem Petri-Netz Editor entworfen wurde, zu laden und auszuführen.

Im Prinzip muss das dotNet-Projekt eine Schnittstelle implementieren und sich bei der Petri-Net Engine als Listener registrieren. Bei der Ausführung wird dann für jede gefeuerte Transition ein Ereignis ausgelöst. Über das ID-Feld der Transition kann die gefeuerte Transition im Ereignis-Handler eindeutig festgestellt werden.

6.3.1. Klassen-Übersicht

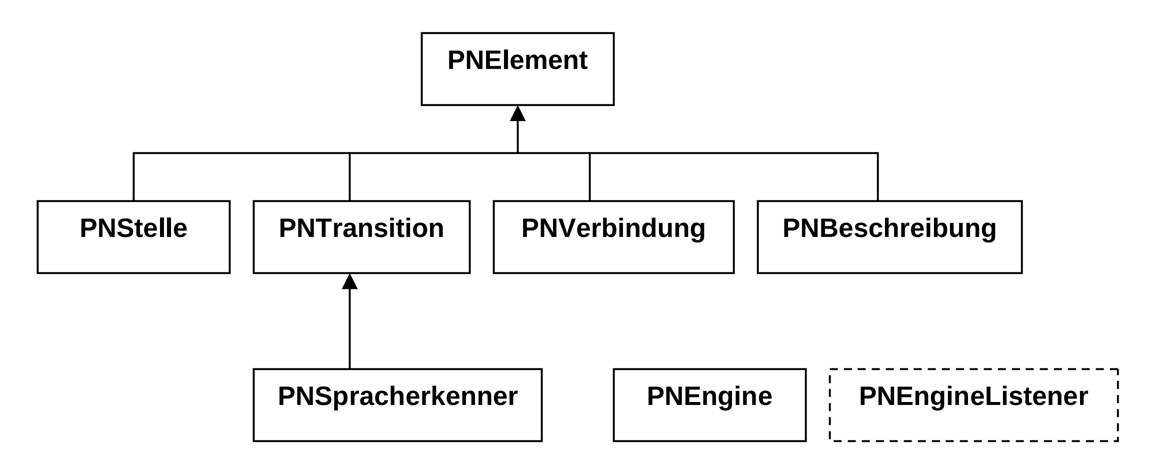


Abb. 33: Klassen-Übersicht PetriNet.dll

6.3.2. PNEngine und PNEngineListener

Zuerst muss eine Instanz der Klasse PNEngine erzeugt, der Listener registriert und das Petri-Netz geladen werden.

```
Imports PetriNet

Public Class …
      Implements PNEngineListener

Dim engine As PNEngine = New PNEngine
engine.addEngineListener(Me)
engine.loadFromFile(“Mein Petri-Netz.ped”)
```

Dann werden die Abbruchbedingungen und sonstigen Einstellungen gesetzt und das Petri-Netz schlussendlich ausgeführt.

```
engine.Abort = False
engine.LogTrace = False
engine.MaxSteps = 100000

engine.doExecute()
```

Die Petri-Net Engine ruft nun für jede gefeuerte Transition die Listener Methoden

```
Public Sub EngineTransitionFiredStart(…)
```

und

```
Public Sub EngineTransitionFiredEnd(…)
```

auf.

Beiden Methoden wird dabei jene Transition übergeben, die gerade gefeuert wird. Da PNTransition von PNElement abgeleitet ist, verfügt die Klasse über das Attribut ID, welche im Petri-Netz Editor für jedes Element vergeben wird und eindeutig sein muss.
Es kann nun mit Hilfe dieser ID eine Select-Anweisung geschrieben werden und derart für jede Transition individuell verzweigt werden.

```
Select Case transition.ID
      Case "lampe1an"
                …
      Case "lampe1aus"
                …
      Case "lampe2an"
                …
      Case "lampe2aus"
                …
End Select
```

Will man vor der Ausführung die Marken von Stellen verändern, so kann man dies durch

```
engine.Marken("operandA") = n
```

Wurden die Marken von Stellen verändert, so sollte vor einer Ausführung unbedingt

```
engine.evalFeuerbereiteTransitionen()
```

aufgerufen werden, um neuerlich alle feuerbereiten Transitionen für den nächsten Schritt zu berechnen.

Mittels

```
n = engine.Marken("ergebnis")
```

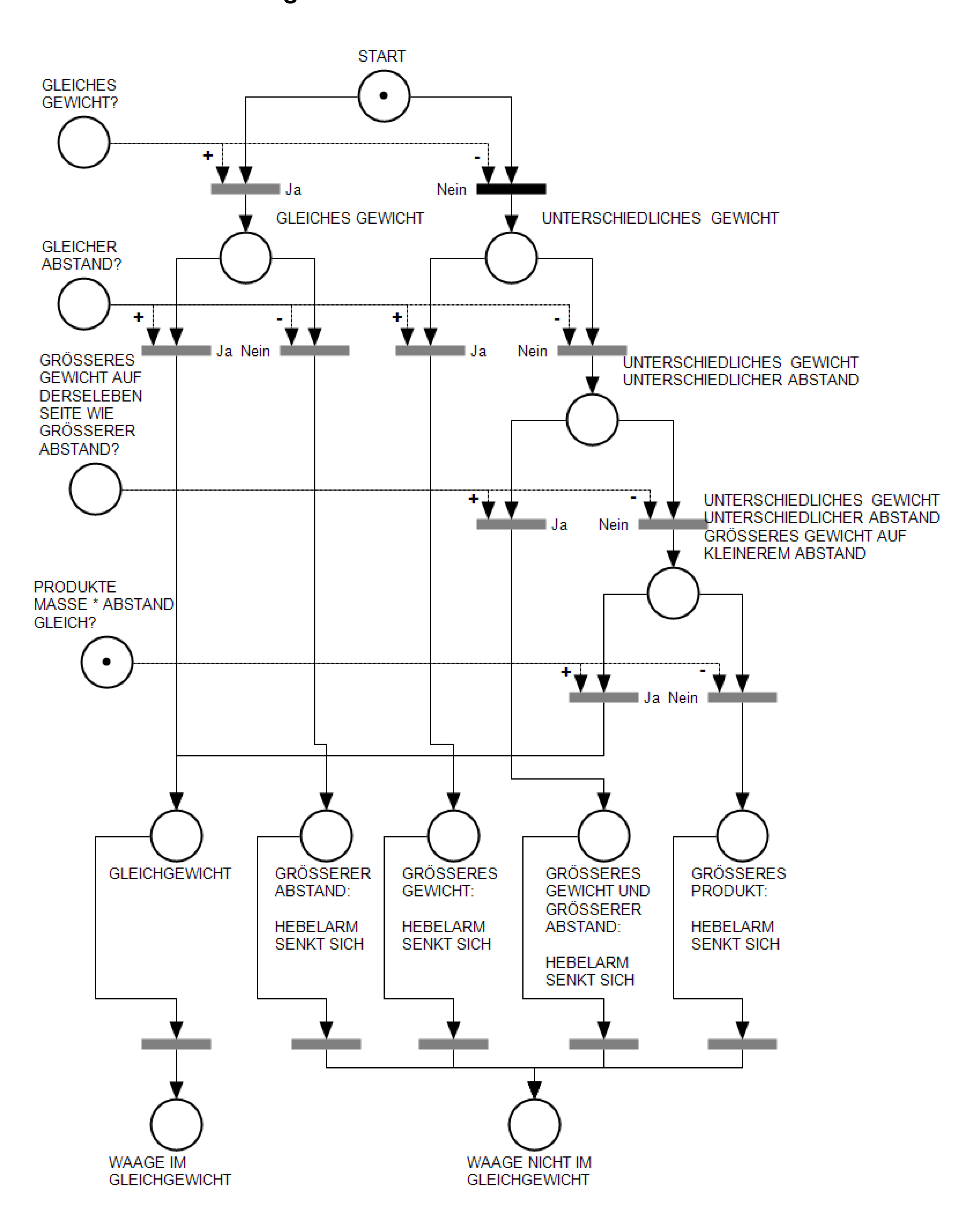
kann die Markenzahl gezielt ausgelesen werden.

6.3.3. Beispiele

6.3.3.1. Lampensteuerung mit Spracheingabe

Siehe Beispiel 6.2.3.1.

6.3.3.2. Entscheidungsnetz für Hebelarm



6.3.4. Beschreibung der Klassen und Schnittstellen

Die zentrale Klasse des Petri-Netz API ist *PNEngine*. Sie verwaltet die Strukturen des Petri-Netzes und ist verantwortlich für das Laden, Ausführen und Speichern. Klassen, die das Interface *PNEngineListener* implementieren und als Listener in der Klasse *PNEngine* registriert sind, werden während der Abarbeitung des Netzes über bestimmte Ereignisse – wie das Feuern einer Transition - informiert.
Die Klasse kann das Petri-Netz auch dynamisch verändern, indem Stellen, Transitionen und Verbindungen hinzugefügt oder gelöscht werden. Marken können verteilt, und die Markenanzahl der Stellen kann abgefragt werden.

Wird ein Petri-Netz durch *PNEngine* gespeichert, kann es nicht wieder im grafischen Petri-Netz Editor geöffnet werden, da sämtliche grafisch relevanten Informationen der Elemente fehlen. Von Pfaden werden beispielsweise die Stützpunkte entfernt, und jede Verbindung auf die wesentlichen Elemente reduziert.
Die im Petri-Netz Editor eingerichteten Lampen eines Petri-Netzes werden während des Ladevorganges ignoriert und müssen über das Listener-Interface implementiert werden.

Alle Strukturelemente sind von der Klasse *PNElement* abgeleitet. Sie unterstützt einen eindeutigen ID-String, einen Text-String, ein anwendungsspezifisches UserData-Objekt und verwaltet in einer Liste alle Verbindungen, an der das Element beteiligt ist. Ihrer Konstruktor-Methode muss jene Instanz von *PNEngine* übergeben werden, welche das Petri-Netz verwaltet. Weiters existieren überschreibbare Methoden, um das Element in einen XmlTextStream zu schreiben und von einem XmlTextStream zu lesen.

PNStelle besitzt zwei weitere Integer-Felder: die Markenanzahl und das Markenlimit.

PNVerbindung besitzt das Feld Typ vom Datentyp *PNVerbindungsTyp* (Normal, Positiv, Negativ), das Integer-Feld Gewicht und die Felder StartElement und EndElement, jeweils vom Datentyp *PNElement*. Dadurch ist auch die Richtung der Verbindung ausgedrückt. Der Konstruktor-Methode muss sowohl jene Instanz der *PNEngine* übergeben werden, die das Petri-Netz verwaltet, als auch die Start- und End-Elemente, mit der diese Verbindung verknüpft ist. Über die connect-Methode wird die Verbindung in die jeweiligen Start- und End-Elemente eingetragen. Mit der disconnect-Methode wird die Verbindung aus den Start- und End-Elementen entfernt.

PNTransition besitzt zwei Listener-Listen - TransitionListenerStart und Transition-ListenerEnd – sowie Methoden, um diese zu verwalten. Wird eine Transition gefeuert, so werden zuerst alle Listener aus TransitionListenerStart informiert. Anschließend wird die doFire-Methode aufgerufen, welche die Markenverteilung ändert. Abschließend werden alle Listener aus TransitionListenerEnd informiert.
PNEngine ist jeweils in beiden Listener-Listen registriert und spielt eine Vermittler-rolle, indem sie diese Ereignisse an das *PNEngineListener*-Interface weiterleitet.
Ein Anwendungsprogramm muss also nur das *PNEngineListener*-Interface implementieren und wird so automatisch über alle gefeuerten Transitionen informiert.

PNSpracherkenner ist von *PNTransition* abgeleitet und besitzt die Infrastruktur, um eine Befehlsliste (SlotList) zu verwalten. Die Befehle sind vom Datentyp *PNSprach-*

erkennerSlot, welcher den Befehls-String, die weiteren Feuerbedingungen für diesen Befehl (startsWith, contains, exact; sowie consume, clear) und den Typ enthält (Normal, Empty, NotRecognized).
Wird ein Spracherkenner gefeuert, so werden zuerst alle Listener aus Transition-ListenerStart informiert. Nun wird der Spracherkenner initialisiert, und die Grammatik entweder erzeugt oder aus einer Datei eingelesen.
Gibt die Funktion canSlotFire True zurück oder rennt der Spracherkenner in ein Timeout, wird fireConnectionsOnSlot mit dem entsprechenden SlotIndex aufgerufen.
Die Befehle haben den Index (SlotIndex) 0 bis n-1 (für n gleich der Anzahl der Befehle), n für den „Leer"-Befehl und n+1 für den „Nicht erkannt"-Befehl.
Die Methode fireConnectionsOnSlot informiert alle Listener aus TransitionListener-End und übergibt zusätzlich den PNSpracherkennerSlot als data-Objekt.
Nachdem alle Slots gefeuert wurden, werden abschließend noch einmal alle Listener aus TransitionListenerEnd informiert - jedoch mit data = nothing.
Mit welchem Slot eine *PNVerbindung* verbunden ist, hängt vom StartSelection-PointIndex bzw. EndSelectionPointIndex ab. Dabei entsprechen Werte von 0 bis 5 jenen 6 Punkten des Spracherkenners, über welche immer gefeuert wird. Die Werte 6 und 7 entsprechen dem „Leer"-Befehl, 8 und 9 dem „Nicht erkannt"-Befehl. Der erste normale Befehl besitzt die Werte 10 und 11, 12 und 13 entsprechen dem zweiten Slot usw.
Allgemein gilt: Normaler SlotIndex = (SelectionPointIndex - 10) / 2 | Integer

PNBeschreibung ist von *PNElement* abgeleitet und enthält keine Erweiterungen. Die Klasse wurde hier nur implementiert, um als mögliche Referenzierung mittels $(…) zu dienen.

6.3.4.1. PNEngine

PNEngineListener-Handler:

```
Public Sub addEngineListener(ByVal l As PNEngineListener)
Public Sub removeEngineListener(ByVal l As PNEngineListener)
Public Sub removeAllEngineListener()

Public Sub fireEngineStartEvent()
Public Sub fireEngineStopEvent()
Public Sub fireEngineAbortEvent()
Public Sub fireEngineStepEvent()
Public Sub fireEngineAudioLevelEvent(ByVal level As Integer)
Public Sub fireEngineRecognizeBufferChangedEvent()
Public Sub fireEngineSpeakBufferChangedEvent()
Public Sub fireEngineLogTraceBeforeStepEvent(ByVal transition As
                                                            PNTransition)
Public Sub fireEngineLogTraceAfterStepEvent(ByVal transition As
                                                            PNTransition)
Public Sub fireEngineTransitionFiredStartEvent(ByVal transition As
                  PNTransition, Optional ByVal data As Object = Nothing)
Public Sub fireEngineTransitionFiredEndEvent(ByVal transition As
                  PNTransition, Optional ByVal data As Object = Nothing)
Public Sub fireEngineProgressBarEvent(ByVal value As Integer)
```

Implementiertes Interface TransitionListener:

```
Public Sub transitionFiredStart(ByVal sender As PNTransition,
                              Optional ByVal data As Object = Nothing)
```

```
Public Sub transitionFiredEnd(ByVal sender As PNTransition,
                              Optional ByVal data As Object = Nothing)
```

Konstruktor und Einstellungen:

```
Public Sub New()
Public Sub createAll()
Public Sub resetStepCounter()
Public Property StepCounter() As Long
Public Property MaxSteps() As Long
Public Property LogTrace() As Boolean
Public Property FileName() As String
Public Property RecognizeBuffer() As String
Public Property SpeakBuffer() As String
Public Property MaxMarkenProStelle() As Integer
Public Property Abort() As Boolean
Public ReadOnly Property FeuerbereiteTransitionen() As ArrayList

Public Sub initialize()
Public Sub clear()
Public Property SnapShot() As ArrayList
```

Zugriff auf Strukturen und deren Attribute:

```
Public Property Marken(ByVal id As String) As Integer
Public Property Limit(ByVal id As String) As Integer

Public Property Gewicht(ByVal id As String) As Integer
Public Property Typ(ByVal id As String) As PNVerbindungsTyp

Public ReadOnly Property SlotList(ByVal id As String) As ArrayList

Public Property Beschriftung(ByVal id As String) As String

Public Function getStelle(ByVal id As String) As PNStelle
Public Function getTransition(ByVal id As String) As PNTransition
Public Function getVerbindung(ByVal id As String) As PNVerbindung
Public Function getBeschriftung(ByVal id As String) As PNBeschreibung
Public Function getElement(ByVal id As String) As PNElement
```

Strukturänderung:

```
Public Sub add(ByVal arr As ArrayList,
                              Optional ByVal isDirty As Boolean = True)
Public Sub add(ByVal element As PNElement,
                              Optional ByVal isDirty As Boolean = True)
Public Sub remove(ByVal arr As ArrayList)
Public Sub remove(ByVal element As PNElement)
```

Datei- und Streamoperationen:

```
Public Sub loadFromFile(ByVal fileName As String)
Public Sub loadFromStream(ByVal reader As XmlTextReader)

Public Sub saveToFile(ByVal fileName As String)
Public Sub saveToStream(ByVal writer As XmlTextWriter)
```

Abarbeitung des Petri-Netzes:

```
Public Overridable Sub doBeforeStart()
Public Overridable Sub doAfterStop()
Public Overridable Sub doAfterAbort()

Public Overridable Function evalFeuerbereitschaft(ByVal t As PNTransition)
                                                                  As Boolean
Public Overridable Sub evalFeuerbereiteTransitionen()
Public Overridable Sub fireTransition(ByVal id As String)
Public Overridable Sub fireTransition(ByVal t As PNTransition)

Public Overridable Sub doExecute()
Protected Sub resumeExecution()

Public ReadOnly Property isExecuting() As Boolean
```

Sonstiges:

```
Public Sub pushAudioProgress(Optional ByVal idx As Integer = -1,
                              Optional ByVal count As Integer = -1)
Public Sub pushProgress(Optional ByVal value As Integer = 0,
                              Optional ByVal total As Integer = 100)

Public Sub showRecognizeBuffer()

Public Overridable Sub clearSpeakBuffer()
Public Overridable Sub doSpeak(ByVal text As String,
                              Optional ByVal waitFor As Boolean = False)
```

6.3.4.2. PNEngineListener

```
Public Sub EngineStart(ByVal sender As PNEngine)
Public Sub EngineStop(ByVal sender As PNEngine)
Public Sub EngineStep(ByVal sender As PNEngine, ByVal stepNr As Long)
Public Sub EngineAbort(ByVal sender As PNEngine)

Public Sub EngineTransitionFiredStart(ByVal sender As PNEngine,
                                    ByVal transition As PNTransition,
                                    Optional ByVal data As Object = Nothing)
Public Sub EngineTransitionFiredEnd(ByVal sender As PNEngine,
                                    ByVal transition As PNTransition,
                                    Optional ByVal data As Object = Nothing)

Public Sub EngineAudioLevel(ByVal sender As PNEngine,
                                    ByVal AudioLevel As Integer)
Public Sub EngineRecognizeBufferChanged(ByVal sender As PNEngine,
                                    ByVal buffer As String)
Public Sub EngineSpeakBufferChanged(ByVal sender As PNEngine,
                                    ByVal buffer As String)

Public Sub EngineLogTraceBeforeStep(ByVal sender As PNEngine,
                                    ByVal transition As PNTransition)
Public Sub EngineLogTraceAfterStep(ByVal sender As PNEngine,
                                    ByVal transition As PNTransition)

Public Sub EngineProgressBar(ByVal sender As PNEngine,
                                    ByVal value As Integer)
```

6.3.4.3. PNElement

Konstruktor und Einstellungen:

```
Public Sub New(ByVal engine As PNEngine)

Public ReadOnly Property Engine() As PNEngine
Public Property ID() As String
Public Property Text() As String
Public Property UserData() As Object
```

Verbindungen:

```
Public ReadOnly Property Connections() As ArrayList

Public Function getInputConnections() As ArrayList
Public Function getOutputConnections() As ArrayList

Public Sub addConnection(ByVal con As PNVerbindung)
Public Sub addConnections(ByVal arr As ArrayList)

Public Sub removeConnection(ByVal con As PNVerbindung)
Public Sub removeConnections(ByVal arr As ArrayList)
Public Sub removeAllConnections()
```

Streamoperationen:

```
Public Overridable Sub loadXML(ByVal reader As System.Xml.XmlTextReader)
Public Overridable Sub saveXML(ByVal writer As System.Xml.XmlTextWriter)
```

Sonstiges:

```
Public Overridable Property ClearSpeakBuffer() As Boolean
Public Property AddToSpeakBuffer() As Boolean
Public Property SpeakAndWaitFor() As Boolean

Public Overridable Sub doSpeak(ByVal text As String,
                              Optional ByVal waitFor As Boolean = False)
```

6.3.4.4. PNStelle

Konstruktor und Einstellungen:

```
Public Sub New(ByVal engine As PNEngine)

Public Property Marken() As Integer
Public Property Limit() As Integer
```

Streamoperationen:

```
Public Overridable Sub loadXML(ByVal reader As System.Xml.XmlTextReader)
Public Overridable Sub saveXML(ByVal writer As System.Xml.XmlTextWriter)
```

6.3.4.5. PNVerbindung

Konstruktoren und Einstellungen:

```
Public Sub New(ByVal engine As PNEngine)
Public Sub New(ByVal engine As PNEngine, ByVal _typ As PNVerbindungsTyp)
Public Sub New(ByVal engine As PNEngine, ByVal startElem As PNElement,
            ByVal endElem As PNElement, ByVal _typ As PNVerbindungsTyp)

Public Property Typ() As PNVerbindungsTyp
Public Property Gewicht() As Integer
Public Property StartElement() As PNElement
Public Property EndElement() As PNElement
Public Property StartSelectionPointIndex() As Integer
Public Property EndSelectionPointIndex() As Integer
```

Verbindungen:

```
Public Sub connect(ByVal startElem As PNElement,
                       ByVal endElem As PNElement)
Public Sub disconnect()
```

Streamoperationen:

```
Public Overridable Sub loadXML(ByVal reader As System.Xml.XmlTextReader)
Public Overridable Sub saveXML(ByVal writer As System.Xml.XmlTextWriter)
```

6.3.4.6. PNTransition

Konstruktor und Einstellungen:

```
Public Sub New(ByVal engine As PNEngine)
```

Streamoperationen:

```
Public Overridable Sub loadXML(ByVal reader As System.Xml.XmlTextReader)
Public Overridable Sub saveXML(ByVal writer As System.Xml.XmlTextWriter)
```

TransitionListenerStart-Handler:

```
Public Sub addTransitionListenerStart(ByVal l As TransitionListener)
Public Sub addTransitionListenerStartAtFirstPos(ByVal l As
                                                      TransitionListener)
Public Sub removeTransitionListenerStart(ByVal l As TransitionListener)
Public Sub removeAllTransitionListenerStart()

Public Function getTransitionListenerStart() As ArrayList

Public Function containsTransitionListenerStart(ByVal l As
                                         TransitionListener) As Boolean

Public Overridable Sub fireTransitionEventStart()
```

TransitionListenerEnd-Handler:

```
Public Sub addTransitionListenerEnd(ByVal l As TransitionListener)
Public Sub addTransitionListenerEndAtFirstPos(ByVal l As
                                                     TransitionListener)
Public Sub removeTransitionListenerEnd(ByVal l As TransitionListener)
Public Sub removeAllTransitionListenerEnd()

Public Function getTransitionListenerEnd() As ArrayList

Public Function containsTransitionListenerEnd(ByVal l As
                                         TransitionListener) As Boolean

Public Overridable Sub fireTransitionEventEnd()
```

Sonstiges:

```
Public Overridable Function canFire() As Boolean
Public Overridable Sub doFireStart()
Public Overridable Function doFire() As Object
```

6.3.4.7. PNSpracherkenner

Konstruktor und Einstellungen:

```
Public Sub New(ByVal engine As PNEngine)

Public Property Timeout() As Integer
Public Property Echo() As Boolean
Public Property AlwaysSpeakPrompt() As Boolean
Public Property ClearRecognitionBuffer() As Boolean
Public Property OnNotRecognizedClearRecognitionBuffer() As Boolean
Public Property GrammerFile() As String
Public ReadOnly Property useGrammerFile() As Boolean
Public ReadOnly Property SlotList() As ArrayList
```

Streamoperationen:

```
Public Overridable Sub loadXML(ByVal reader As System.Xml.XmlTextReader)
Public Overridable Sub saveXML(ByVal writer As System.Xml.XmlTextWriter)
```

Sonstiges:

```
Public Overrides Function canFire() As Boolean
Public Overrides Sub doFireStart()
Public Overrides Function doFire() As Object

Public Function canSlotFire() As Boolean
Public Sub fireConnectionsOnSlot(ByVal idx As Integer)

Public Sub initSpracherkennung()

Public Overridable Sub consumeRecognized(
                       ByVal slot As PNSpracherkennerSlot,
                       Optional ByVal firstOccurence As Boolean = True)
Public Overridable Sub clearRecognizedBuffer()
```

6.3.4.8. PNBeschreibung

Konstruktor und Einstellungen:

```
Public Sub New(ByVal engine As PNEngine)
```

Streamoperationen:

```
Public Overridable Sub loadXML(ByVal reader As System.Xml.XmlTextReader)
Public Overridable Sub saveXML(ByVal writer As System.Xml.XmlTextWriter)
```

6.4. Mathematische Grundlagen

Definition 6.4.1. Petri-Netz.
Ein Petri-Netz N ist ein Vier-Tupel N = (P, T, I, O):

- $P = \{ p_1, p_2, \ldots, p_n \}$ ist eine endliche Menge von Stellen, $n \geq 0$.
- $T = \{ t_1, t_2, \ldots, t_m \}$ ist eine endliche Menge von Transitionen, $m \geq 0$.
- I ist die Input-Funktion, eine Abbildung ($I: T \rightarrow P^*$) von Transitionen auf Multimengen von Stellen (Zuordnung der Eingangsstellen).
- O ist die Output-Funktion, eine Abbildung ($O: T \rightarrow P^*$) von Transitionen auf Multimengen von Stellen (Zuordnung der Ausgangsstellen).

Die Menge der Stellen und der Transitionen sind disjunkt, $P \cap T = \varnothing$. ♦

Definition 6.4.2. Markierung.
Die Markierung μ eines Petri-Netzes N = (P, T, I, O) ist eine Abbildung der Menge der Stellen P auf die nicht negativen, ganzen Zahlen N_0 ($\mu: P \rightarrow N_0$).
Die Markierung μ wird definiert als ein n-dimensionaler Vektor $\mu = (\mu_1, \mu_2, \ldots, \mu_n)$, wobei $n = |P|$ gilt, und jedes $\mu_i \in N_0$ ist (für i = 1, 2, …, n).
Der Vektor μ gibt dann für jede Stelle p_i an, wie viele Marken μ_i in dieser Stelle sind (für i = 1, 2, …, n).
Die oben erwähnte Funktion ist offensichtlich:

$$\mu(p_i) = \mu_i \quad (i = 1, 2, \ldots, n) \tag{6.4.1}$$

♦

Definition 6.4.3. Anzahlfunktion.
Die Funktion $\#(p_i, I(t_j))$ gibt an, wie viele Kanten von $p_i \in P$ nach $t_j \in T$ gerichtet sind:

$$\#(p_i, I(t_j)) = \begin{cases} 0, & \text{falls } p_i \notin I(t_j), \\ \text{Anzahl der Elemente } p_i \in I(t_j) & \text{sonst,} \end{cases} \tag{6.4.2}$$

bzw. wie viele Kanten von t_j nach p_i führen:

$$\#(p_i, O(t_j)) = \begin{cases} 0, & \text{falls } p_i \notin O(t_j), \\ \text{Anzahl der Elemente } p_i \in O(t_j) & \text{sonst.} \end{cases} \tag{6.4.3}$$

♦

Definition 6.4.4. Kantengewicht.
Ist $n = \#(p_i, I(t_j)) \neq 0$, so sind Kanten von $p_i \in P$ nach $t_j \in T$ gerichtet. Diese n Kanten können in einer Kante mit dem *Kantengewicht* n zusammengefasst werden.
Ist hingegen $m = \#(p_i, O(t_j)) \neq 0$, so führen m Kanten von $t_j \in T$ nach $p_i \in P$. Diese m Kanten können ebenfalls in einer Kante mit dem *Kantengewicht* m zusammengefasst werden. ♦

Definition 6.4.5. Schaltbedingung.
Eine Transition $t_j \in T$ in einem markierten Petri-Netz $N_m = (P, T, I, O, \mu)$ heißt aktiviert, wenn für alle $p_i \in P$ gilt:

$$\mu(p_i) \geq \#(p_i, I(t_j)) \tag{6.4.4}$$

♦

Definition 6.4.6. „Feuern“ einer Transition.
Ist die Schaltbedingung für eine Transition $t_j \in T$ in einem gegebenen markierten Petri-Netz $N_m = (P, T, I, O, \mu)$ erfüllt, d.h., ist sie aktiviert, so ist das Feuern dieser Transition der Übergang von der Markierung μ zu μ', die folgendermaßen definiert ist:

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j)) \tag{6.4.5}$$

für alle $p_i \in P$. ♦

Definition 6.4.7. Next-state-Funktion.
Die Next-state-Funktion $\delta(\mu, t_j)$ ist für ein markiertes Petri-Netz Nm = (P, T, I, O, μ) mit der *Markierung* μ und der *Transition* $t_j \in T$ dann und nur dann definiert, wenn gilt

$$\mu(p_i) \geq \#(p_i, I(t_j)) \tag{6.4.6}$$

für alle $p_i \in P$ (d.h., t_j ist aktiviert). Wenn $\delta(\mu, t_j)$ definiert ist, dann gilt:

$$\delta(\mu, t_j) = \mu' \tag{6.4.7}$$

mit

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j)) \tag{6.4.8}$$

für alle $p_i \in P$. ♦

Definition 6.4.8. Reachability-Set.
Das *Reachability-Set* $R(N_m)$ für ein markiertes Petri-Netz $N_m = (P, T, I, O, \mu_0)$ mit der Anfangsmarkierung μ_0 ist die kleinste Menge der Markierungen, die sich folgendermaßen zusammensetzt:

1. $\mu_0 \in R(N_m)$;
2. falls $\mu \in R(N_m)$ und weiters für eine beliebige Transition $t_j \in T$ gilt $\mu' = \delta(\mu, t_j)$, dann ist auch $\mu' \in R(N_m)$. ♦

Definition 6.4.9. Relation f .
Eine Markierung μ' ist einer Markierung μ ähnlich (μ' **f** μ), falls folgende Relationen zutreffen:

$$\mu'(p_i) \geq \mu(p_i) \text{ für alle Stellen } p_i \in P \text{, und}$$
$$\mu'(p_j) > \mu(p_j) \text{ für mindestens eine Stelle } p_j \in P. \tag{6.4.9}$$

♦

Definition 6.4.10. Positive Kommunikationslinie.
Die Transition kann nur feuern, wenn in der Eingangsstelle p mindestens eine Marke vorhanden ist. Über die Kommunikationslinie erfolgt ein Anstoß an die Transition t, jedoch ohne dass aus der Stelle p eine Marke abgezogen wird. Positive Kommunikationslinien werden durch strichlierte Kanten dargestellt, die durch ein „+" gekennzeichnet werden. ♦

Definition 6.4.11. Negative Kommunikationslinie.
Die Transition kann nur feuern, wenn in der Eingangsstelle p *keine* Marke vorhanden ist. Auch in diesem Fall erfolgt kein Markentransport, sondern nur eine Aktivierung der Transition t. Negative Kommunikationslinien werden durch strichlierte Kanten dargestellt, die durch ein „-" gekennzeichnet werden. ♦

Definition 6.4.12. Erweiterte Anzahlfunktion.
Die Funktion $\#(p_i, I(t_j))$ gibt an, wie viele Kanten von $p_i \in P$ nach $t_j \in T$ gerichtet sind:

$$\#(p_i, I(t_j)) = \begin{cases} 0, & \text{falls } p_i \notin I(t_j), \\ +, & \text{für eine positive Kommunikationslinie}, \\ -, & \text{für eine negative Kommunikationslinie}, \\ \text{Anzahl } p_i \in I(t_j) & \text{sonst.} \end{cases} \tag{6.4.10}$$

♦

Definition 6.4.13. Stellenkapazität.
Die *Stellenkapazität* K einer Stelle gibt an, wie viele Marken diese Stelle maximal aufnehmen kann, wodurch die Stelle K-*bounded* wird. Sind in einer Stelle mit der Kapazität K bereits k Marken enthalten, so kann die Eingangstransition nicht mehr feuern, und die Schaltbedingung ist nicht mehr erfüllt. ♦

Definition 6.4.14. Erweiterte Schaltbedingung.
Eine Transition $t_j \in T$ in einem markierten Petri-Netz $N_m = (P, T, I, O, \mu)$ heißt *aktiviert*, wenn für alle $p_i \in P$ mit der Kapazität K_i gilt:

$$\mu(p_i)=\begin{cases} \geq n \text{ und } \mu(p_i)-\#(p_i,I(t_j))+\#(p_i,O(t_j))\leq K_i, & \text{wenn } \#(p_i,I(t_j))=n, \\ \geq 1 \text{ und } \mu(p_i)+\#(p_i,O(t_j))\leq K_i, & \text{wenn } \#(p_i,I(t_j))=+, \\ =0 \text{ und } \mu(p_i)+\#(p_i,O(t_j))\leq K_i, & \text{wenn } \#(p_i,I(t_j))=-. \end{cases} \quad (6.4.11)$$

♦

Definition 6.4.15. Erweiterte Schaltregel.
Ist die Schaltbedingung für eine Transition $t_j \in T$ in einem gegebenen markierten Petri-Netz $N_m = (P, T, I, O, \mu)$ erfüllt, d.h., ist sie aktiviert, so ist das Feuern dieser Transition der Übergang von der Markierung μ zu μ', die folgendermaßen definiert ist:

$$\mu'(p_i)=\begin{cases} \mu(p_i)-\#(p_i,I(t_j))+\#(p_i,O(t_j)), & \text{wenn } \#(p_i,I(t_j))=n, \\ \mu(p_i)+\#(p_i,O(t_j)), & \text{wenn } \#(p_i,I(t_j))=+, \\ \mu(p_i)+\#(p_i,O(t_j)), & \text{wenn } \#(p_i,I(t_j))=-. \end{cases} \quad (6.4.12)$$

♦

Definition 6.4.16. Erweiterte Next-state-Funktion.
Die Next-state-Funktion $\delta(\mu, t_j)$ ist für ein markiertes Petri-Netz $N_m = (P, T, I, O, \mu)$ mit der *Markierung* μ und der *Transition* $t_j \in T$ dann und nur dann für alle *Stellen* $p_i \in P$ mit der *Kapazität* K_i definiert, wenn gilt:

$$\mu(p_i)=\begin{cases} \geq n \text{ und } \mu(p_i)-\#(p_i,I(t_j))+\#(p_i,O(t_j))\leq K_i, & \text{wenn } \#(p_i,I(t_j))=n, \\ \geq 1 \text{ und } \mu(p_i)+\#(p_i,O(t_j))\leq K_i, & \text{wenn } \#(p_i,I(t_j))=+, \\ =0 \text{ und } \mu(p_i)+\#(p_i,O(t_j))\leq K_i, & \text{wenn } \#(p_i,I(t_j))=-. \end{cases} \quad (6.4.13)$$

Wenn die Transition t_j aktiviert ist, und $\delta(\mu, t_j)$ definiert ist, dann gilt $\delta(\mu, t_j) = \mu'$ mit

$$\mu'(p_i)=\begin{cases} \mu(p_i)-\#(p_i,I(t_j))+\#(p_i,O(t_j)), & \text{wenn } \#(p_i,I(t_j))=n, \\ \mu(p_i)+\#(p_i,O(t_j)), & \text{wenn } \#(p_i,I(t_j))=+, \\ \mu(p_i)+\#(p_i,O(t_j)), & \text{wenn } \#(p_i,I(t_j))=-. \end{cases} \quad (6.4.14)$$

Für alle $p_i \in P$. ♦

6.5. Abbildungen

Abb. 1: Petri-Netz Editor starten
Abb. 2: Komponenten des Petri-Netz Editors
Abb. 3: Kontextmenü
Abb. 4: Menü *Datei*, *Bearbeiten, Petri-Netz, Fenster* und *Hilfe*
Abb. 5: Werkzeugleisten mit Datei-, Bearbeitungs-, Entwurfs- und Simulatorfunktionen
Abb. 6: StatusZeile
Abb. 7: Einstellungen
Abb. 8: Druckvorschau
Abb. 9: Lampenanzeige
Abb. 10: Kontextmenü: Erweiterungsbereich und Lampe
Abb. 11: Lampeneinstellungen
Abb. 12: Ansichten einer Stelle – normal und selektiert
Abb. 13: Darstellungen einer Stelle (Markenzahl, Beschränkung)
Abb. 14: *Eigenschaften* einer Stelle
Abb. 15: Ansichten einer Transition – normal und selektiert, je horizontal und vertikal
Abb. 16: *Eigenschaften* einer Transition
Abb. 17: Ansichten eines Spracherkenners – normal und selektiert
Abb. 18: *Eigenschaften* eines Spracherkenners
Abb. 19: Dialog *Sprachbefehle* eines Spracherkenners
Abb. 20: Ansichten einer Verbindung – normal und selektiert
Abb. 21: Positive und negative Kommunikationslinie
Abb. 22: Pfad mit einem Kantengewicht > 1
Abb. 23: *Eigenschaften* einer Kommunikationslinie
Abb. 24: *Eigenschaften* einer Verbindung
Abb. 25: Beispiel für eine Beschriftung
Abb. 26: *Eigenschaften* einer Beschriftung
Abb. 27: Werkzeugleiste mit Simulatorfunktionen
Abb. 28: Knopf für die schrittweise Ausführung
Abb. 29: SAPI – Die wichtigsten Elemente und Attribute
Abb. 30: Eigenschafts-Bereich einer Transition mit den Optionen für die Sprachsynthese
Abb. 31: Referenzierung
Abb. 32: Grammar – Die wichtigsten Elemente und Attribute
Abb. 33: Klassen-Übersicht PetriNet.dll

6.6. Literatur

[MS Speech SDK] Microsoft Speech SDK (SAPI 5.1) – Hilfedatei. Microsoft Corporation, 1995 - 2004
[PED] Oprea, Valentin: Petri-Netz Editor (PED) Benutzerhandbuch. Wien, 2002
[PN] Bernd Rosenstengel; Udo Winand: Petri-Netze. Braunschweig/Wiesbaden, Vieweg Verlag, 1991
[PROZAUT] Kastner, W.; Schild, G.-H.: Prozeßautomatisierung. Wien: Springer Verlag, 1998